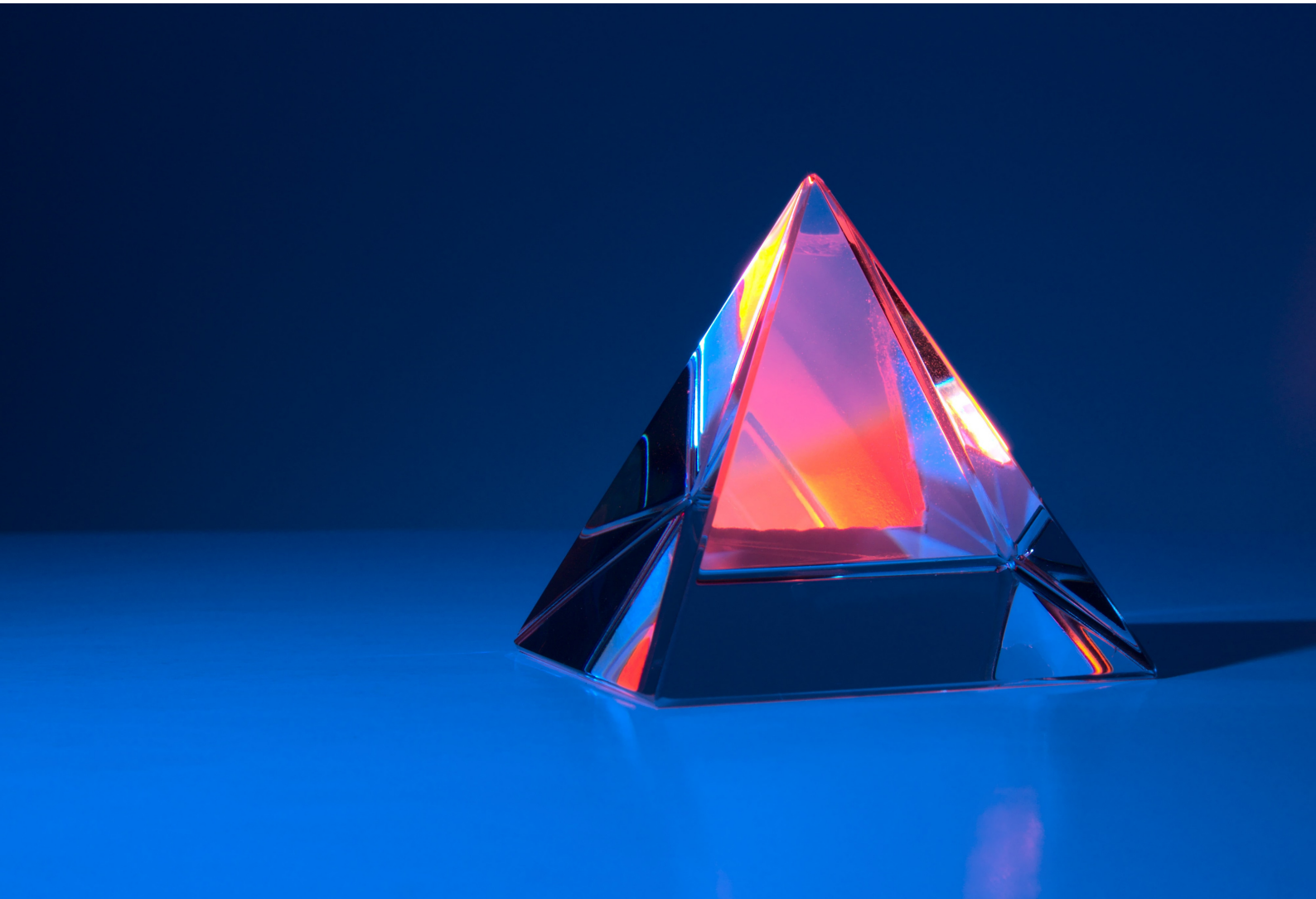
Krystyna Ślusarczyk

C#/.NET

50 ESSENTIAL INTERVIEW QUESTIONS

Junior Level



HELLO!

This e-book is a part of my course "C#/.NET - 50 Essential Interview Questions (**Junior Level**)".

https://bit.ly/3hSRpOq

You can find every lecture from the course here.

You can also check out the full course "C#/.NET - 50 Essential Interview Questions (**Mid Level**)" which you can find under this link:

https://bit.ly/3sC7FsW

INTRODUCTION

Hello, I'm Krystyna! I'm a programmer who loves to write elegant code.
I've been working as a software developer since 2013. About half of this time I've been engaged in teaching programming.
I believe that with a proper explanation, everyone can understand even the most advanced topics related to programming.
I hope I can show you how much fun programming can be, and that you will enjoy it as much as I do!

CONTENTS

16.What is a static class?
17.What is the purpose of the ternary conditional operator?
18.What is the purpose of the null coalescing and null conditional operators?
19.What is encapsulation?
20.What is LINQ?
21.What are extension methods?
22.What is IEnumerable?
23.What is the difference between the equality operator (==) and Equals?
24.What is the difference between deep copy and shallow copy?
25.What is the Garbage Collector?
26.What are nullable types?
27.What is a property?
28.What are generics?
29.What is the difference between the "const" and the "readonly" modifiers?
30.What is the difference between the "ref" and the "out" keywords?
31.What is the difference between an interface and an abstract class?
32.What is polymorphism?
33.What's the difference between a virtual method and an abstract method?
34.What is the method overloading?
35.What is the difference between method overriding and method hiding?
36.Does C# support multiple inheritance?

37.What is the DRY principle?
38.What is the "magic number" antipattern?
39.Why is using the "goto" keyword considered a bad practice?
40.What is the "spaghetti code"?
41.What is the Singleton design pattern?
42.What is the Builder design pattern?
43.What is the Adapter design pattern?
44.What is the Bridge design pattern?
45.What is the Factory Method design pattern?
46.What is the "S" in the SOLID principles?
47.What is the "O" in the SOLID principles?
48.What is the "L" in the SOLID principles?
49.What is the "I" in the SOLID principles?
50.What is the "D" in the SOLID principles?

1. What is the Common Intermediate Language?

Brief summary: The Common Intermediate Language is a programming language that all .NET-compatible languages like C#, Visual Basic, or F# get compiled to.

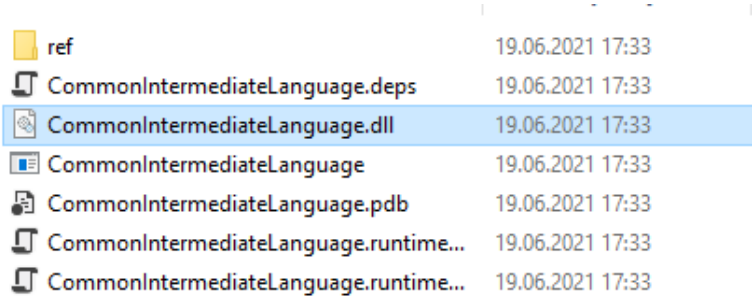
The **Common Intermediate Language** (sometimes, for short, referred to as the Intermediate Language) is a programming language. When the source code written in .NET-compatible languages like C#, Visual Basic, or F# gets compiled, it is transformed into code written in Common Intermediate Language. When the application is started, the Common Language Runtime's **JIT Compiler** translates the CIL code to binary code (JIT stands for "Just-In-Time", which means that a particular piece of code will be translated from CIL to binary code just before it is executed for the first time).

Let's write a simple C# code and see what it looks like after being translated to CIL:

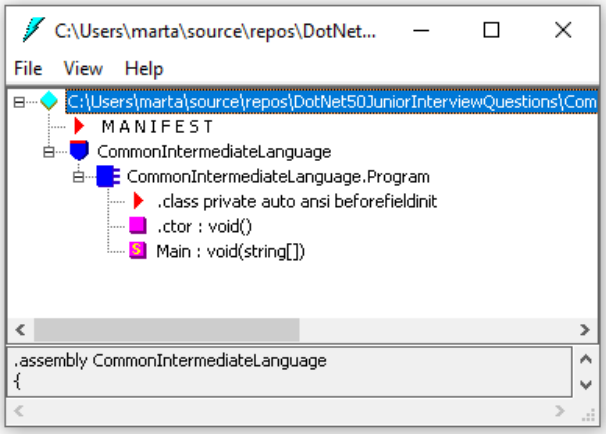
```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello! What's your name?");
        var name = Console.ReadLine();

        Console.WriteLine($"Nice to meet you, {name}. How old are you?");

        var ageAsText = Console.ReadLine();

        if(int.TryParse(ageAsText, out int age))
        {
            Console.WriteLine($"That young, only {age}?");
        }
        else
        {
            Console.WriteLine("Sorry, I didn't get that.");
        }
        Console.WriteLine("Well, it was nice to meet you! Bye, bye!");
        Console.ReadKey();
    }
}
```

To see the CIL code, we must first make sure to build the solution in the Visual Studio. Then we are going to use **Ildasm** to view the CIL code. Ildasm is the Intermediate Language Disassembler, and it gets installed when you install .NET on your machine. On my machine it got installed in C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\ildasm.exe. Let's run this tool:

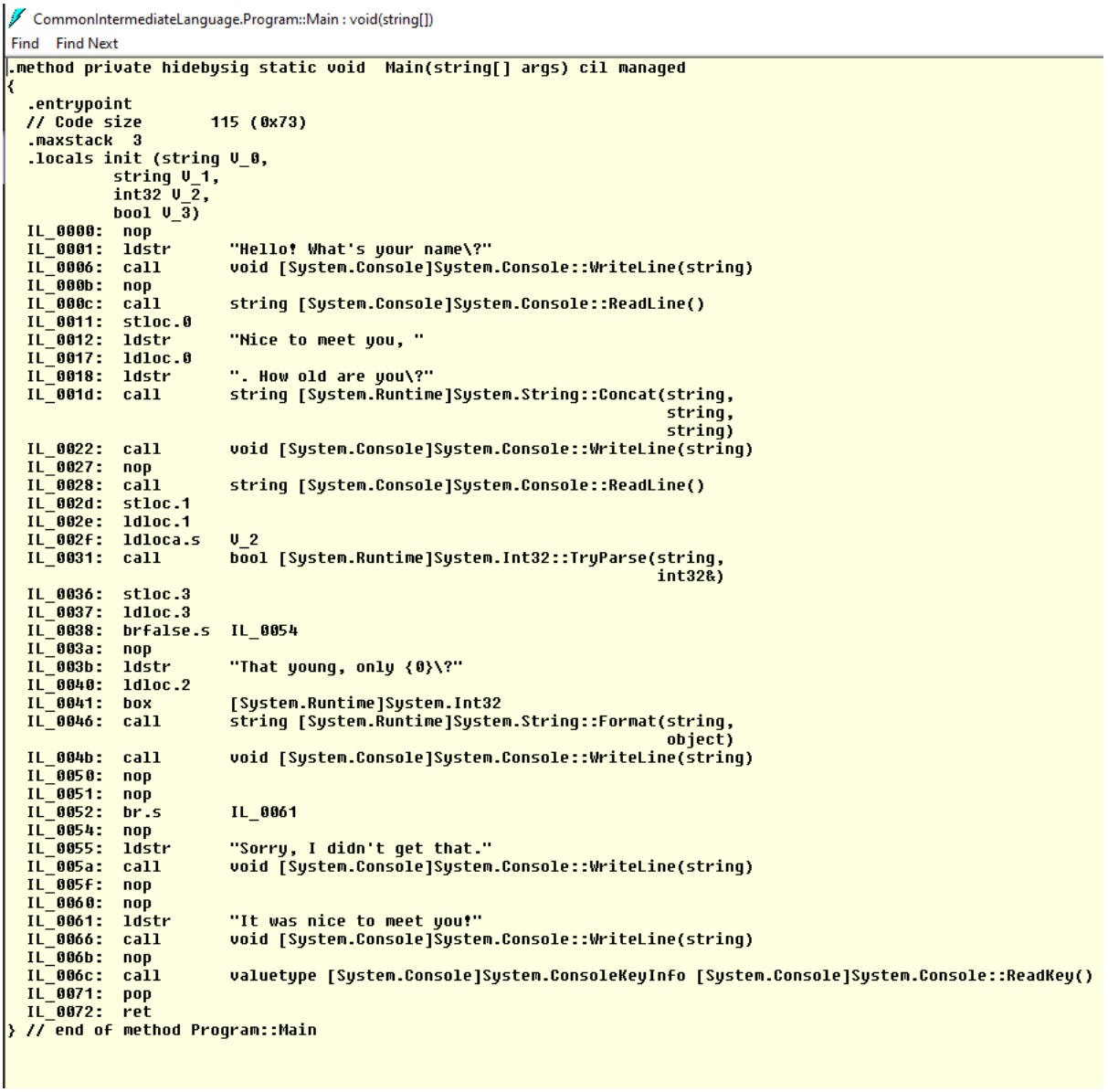
Now, we must find the *.dll for which we want to see the CIL code. The simplest solution is to right-click the project in the Visual Studio and then select "Open Folder in File Explorer". Then, we must go to the output folder. In my case it is /bin/Debug/net5.0. There I can find the *.dll that was built by the Visual Studio:



We can simply drag and drop it to Ildasm:



All right, let's see how the Main method looks in CIL code. Let's double-click on the Main method in Ildasm. This is what we will see:



```
CommonIntermediateLanguage.Program::Main : void(string[])
Find  Find Next
.method private hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       115 (0x73)
  .maxstack  3
  .locals init (string V_0,
           string V_1,
           int32 V_2,
           bool V_3)
  IL_0000:  nop
  IL_0001:  ldstr      "Hello! What's your name\?"
  IL_0006:  call       void [System.Console]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  call       string [System.Console]System.Console::ReadLine()
  IL_0011:  stloc.0
  IL_0012:  ldstr      "Nice to meet you, "
  IL_0017:  ldloc.0
  IL_0018:  ldstr      ". How old are you\?"
  IL_001d:  call       string [System.Runtime]System.String::Concat(string,
                                                                    string,
                                                                    string)
  IL_0022:  call       void [System.Console]System.Console::WriteLine(string)
  IL_0027:  nop
  IL_0028:  call       string [System.Console]System.Console::ReadLine()
  IL_002d:  stloc.1
  IL_002e:  ldloc.1
  IL_002f:  ldloca.s   V_2
  IL_0031:  call       bool [System.Runtime]System.Int32::TryParse(string,
                                                                   int32&)
  IL_0036:  stloc.3
  IL_0037:  ldloc.3
  IL_0038:  brfalse.s  IL_0054
  IL_003a:  nop
  IL_003b:  ldstr      "That young, only {0}\?"
  IL_0040:  ldloc.2
  IL_0041:  box        [System.Runtime]System.Int32
  IL_0046:  call       string [System.Runtime]System.String::Format(string,
                                                                    object)
  IL_004b:  call       void [System.Console]System.Console::WriteLine(string)
  IL_0050:  nop
  IL_0051:  nop
  IL_0052:  br.s       IL_0061
  IL_0054:  nop
  IL_0055:  ldstr      "Sorry, I didn't get that."
  IL_005a:  call       void [System.Console]System.Console::WriteLine(string)
  IL_005f:  nop
  IL_0060:  nop
  IL_0061:  ldstr      "It was nice to meet you!"
  IL_0066:  call       void [System.Console]System.Console::WriteLine(string)
  IL_006b:  nop
  IL_006c:  call       valuetype [System.Console]System.ConsoleKeyInfo [System.Console]System.Console::ReadKey()
  IL_0071:  pop
  IL_0072:  ret
} // end of method Program::Main
```

This might not be the most beautiful programming language you've ever seen, but on the other hand, it is not completely unreadable, as one could expect.

Remember - all .NET compatible languages, not only C#, get compiled to the CIL. That enables communication between, for example, a C# and an F# libraries. For example, we can have a C# class derived from an F# class exactly because they both get compiled to the same programming language - the CIL.

Tip: other interview questions on this topic:

- "**How is it possible that a C# class can derive from, for example, an F# class?**"
  It is possible because both those languages are .NET compatible and they get compiled to the Common Intermediate Language.

- "**Does C# compiler compile C# source code directly to binary code?**"
  No, it compiles it to the Intermediate Language, which is compiled to binary code by the Just-In-Time compiler in runtime.

- "**How can you see the CIL code a project got compiled to?**"
  *Some tools can decompile a *.dll file and read the CIL code. One of those tools is Ildasm.*

- *"***What is the Just-In-Time compiler?***"*
  Just-In-Time compiler is a feature of the Common Language Runtime (CLR), which translates the Common Intermediate Language (CIL) code to binary code during the program execution.

2. What is the Common Language Runtime (CLR)?

Brief summary: The Common Language Runtime is a runtime environment that manages the execution of the .NET applications.

The **Common Language Runtime** is a runtime environment that manages the execution of .NET applications. The CLR works as a special "operating system" for .NET applications, that manages all operations (like memory management) that otherwise must have been dealt with by a programmer. The CLR stands between the actual operating system (for example Windows) and the application.

Before we move on, I would like to say one thing: you may think that CLR is not a junior-level topic, and you might be right - this is a pretty low-level feature of .NET and you certainly can start programming .NET applications without even knowing that the CLR exists. Nevertheless, I wanted to introduce this topic as throughout this course I mention the CLR very often, as it affects many aspects of .NET programming. I want to make sure every subject in this course is explained in detail, even if it sometimes exceeds the junior level. **Some of the topics of this course simply can't be understood thoroughly without a basic understanding of the role of the CLR.**

All right, let's move on then. The important thing to understand is that the CLR is the .NET component that is not exclusive for C# applications. All programs written for the .NET are executed by the CLR. All code executed under the CLR is called the **managed code**. Thanks to the CLR, cross-language integration is supported in .NET. For example, you can have a C#'s class derived from a class defined in F#, because the CLR can understand both languages (because both are compiled to the Intermediate Language).

The CLR is responsible for many operations essential for any .NET application to work. Some of them are:

- **JIT (Just-in-time) compilation** - the compilation of the Common Intermediate Language to the binary code. Thanks to that the .NET applications can be used cross-platform because the code is compiled to platform-specific binary code only right before execution. See the "What is the Common Intermediate Language (CIL)?" lecture for more information on that.

- **Memory management** - CLR allocates the memory needed for every object created within the application. CLR also includes the Garbage Collector, which is responsible for releasing and defragmenting the memory. See the "What is the Garbage Collector?" lecture for more information.
- **Exception handling** - when the exception is thrown, the CLR makes sure the code execution is redirected to the proper catch clause. See the "How are exceptions handled in C#?" lecture for more information.
- **Thread management** - threads are beyond junior level, so let's just shortly say that the CLR manages the execution of the multi-threaded applications, making sure all threads work together well
- **Type safety** - part of the CLR is the **CTS - Common Type System**. CTS defines the standard for all .NET-compatible languages. Thanks to that, the CLR can understand types defined in C#, F#, Visual Basic, and so on, enabling cross-language integration.
- And many more

The CLR is the implementation of the CLI - **Common Language Infrastructure**. CLI was originally created by Microsoft and is standardized by ISO and ECMA. Sounds confusing? Let's explain it in this way - CLI is like a design of a house - it describes where walls, windows, piping, and electricity goes. It was originally created by Microsoft. ISO and ECMA are like state authorities who approve the design and make sure it is safe and reasonable, and that houses built based on this design will all function properly. Using this design, Microsoft builds a house. In this metaphor, the design is the CLI and the house built by Microsoft is the CLR. The thing about designs is that we can build many things based on the same design. That means another company could implement its own version of the project in accordance with the CLI. Does it ever happen? Actually, it does! For example, there is Mono Runtime, a counterpart of the CLR developed by a company called Ximian.

By now you should have a general idea of what the CLR does, but before we wrap up let's go **step-by-step** through a (simplified) process of creating and running the application to see when and how the CLR plays its role.

1. The programmer writes the program, which at first is just a bunch of text files.
2. The compiler compiles the text file to Common Intermediate Language, which is platform-independent. The compiler also prepares the metadata that describes all the types along with the methods they include.
3. Now the CLR comes into play. It starts the program under the specific operating system.
4. The CLR's Just-In-Time compiler compiles the Intermediate Language to binary code that can be interpreted by the machine's operating system. It uses the metadata prepared by the compiler.

5. As the application runs, the CLR manages all its low-level aspects - memory management, threads, exceptions handling, and so on.
6. When the application stops, the CLR's job is done.

As you can see, the **CLR is the critical component of .NET**. It manages basically everything that happens under the hood of the running application, allowing us, the programmers, to focus on business aspects of the development. Before tools like the CLR were introduced, programmers must have dealt with all things like memory management, which is the case in languages like C (can you imagine allocating memory for the objects by hand?).

Tip: other interview questions on this topic:

- "**What is the difference between CLR, CLI, and CIL?**"
  CLR (Common Language Runtime) is an implementation of the CLI (Common Language Infrastructure). CIL is Common Intermediate Language, to which all .NET-compatible languages get compiled.

- "**What is the CTS?**"
  CTS is the Common Type System, which is a standardized type system for all .NET-compatible languages, which makes them interoperable - for example, we can have a C# class derived from an F# class.

- "**Is the CLR the only implementation of the CLI**?"
  No. Anyone can create their implementation of the CLI. One of the examples is Mono Runtime.

3. What is the difference between C# and .NET?

Brief summary: C# is a programming language and .NET is a framework that supports applications written in C#, as well as in other .NET compatible languages.

The difference between C# and .NET is a common source of confusion. When browsing through job offers, you might encounter job titles like:

- "C# Developer"
- ".NET Developer"
- "C#/.NET Developer"
- ".NET Developer with C#"

No wonder people tend to consider C# and .NET synonyms. But they are not! Let's see what the difference actually is:

- **C#** is a **programming language**. Nothing more. If you want, you can develop a C#'s compiler that will translate the *.cs file into a binary code that can be run at any platform without using .NET. However, Microsoft's implementation of C# is heavily integrated with .NET, and in almost every practical case applications written with C# will be run under .NET.
- **.NET** is a **framework** that enables running of applications written in C# and other languages compatible with it (like F# or Visual Basic). You can think of it like that: **C# is a plane**, and **.NET is an airport**. One is not very useful without the other - you can't really use the plane without all the airport's infrastructure, like fueling, runway, control towers, passenger or cargo access. On the other hand, an airport without a plane is just an expensive mall really far from the city center.

  So what is the **role of .NET**, exactly?
  - It provides the **execution environment** called Common Language Runtime, which is responsible for things like processing the Intermediate Language, managing the memory, providing error handling, and more. You can learn more about the CLR in the "What is the Common Language Runtime (CLR)?" lecture
  - It provides a set of **standard libraries** (which can be found in the System namespace)

Let's get back to the job titles you might encounter. Employers are sometimes reluctant to post job offers for "C# developer" because they want to hire people who understand the .NET in general, possibly know some .NET-related technologies (like WPF, MVC or Entity Framework). On the other hand, it doesn't make much sense to post a job offer for ".NET Developer" if one actually means "C# programmer", because F# or Visual Basic programmers are also working in .NET, so they could eagerly answer this job offer, and that would lead to a pretty awkward interview.

By the way, at this point you might be curious **why ".NET" and ".NET Framework" are also used interchangeably**. I admit - this is actually confusing. First, there was the .NET Framework, released in 2002. The truth is, people rarely used the full name, and everyone was calling it just ".NET". Then, in 2016, .NET Core - a successor of .NET Framework - was released. So at this point we had ".NET Framework" which was commonly called ".NET", and ".NET Core". The real problem started in 2020 - when Microsoft released version 5.0 of ".NET Core" but actually decided that from now on, this technology will be called ".NET". So nowadays, when people say ".NET" they are often asked "But do you mean the old .NET Framework or the latest .NET Core?". Well, if you expected the programming to be simple, you were wrong.

Since throughout this course I'm using .NET 5.0 (so the 2020 version of the technology once called .NET Core) I'm going to use the ".NET" name.

Tip: other interview questions on this topic:

- "**What is the difference between .NET and .NET Framework?**" *.NET is a successor of .NET Framework. .NET was originally named .NET Core, and it was renamed to .NET since version 5.0.*

4. What is the difference between value types and reference types?

Brief summary: The differences between value types and reference types are:

1. Value types inherit from System.ValueType while reference types inherit from System.Object.
2. When a value type is passed as a parameter, its copy is given to the method. When a reference type is passed as a parameter, a copy of the reference is given to the method.
3. On assignment, a variable of a value type is copied. For reference types, only a reference is copied.
4. All value types are sealed (which means, they cannot be inherited)
5. Value types are stored on the stack, reference types are stored on the heap (because of that, the Garbage Collector only cleans up reference types)

In C# we distinguish **two** types of variables:

- First are **value types**. Simple built-in types like int, double, DateTime, bool are value types. Also, all structs are value types.
- Second are **reference types**. Object, String, StringBuilder, List, Array, HttpClient, XmlSerializer, and all user-defined classes are reference types.

The fundamental difference between them is that a reference type variable only holds a reference (you can think of it as a link or an address) to the actual data, while the value type variable holds the actual data.

To better understand this let's consider two real-life scenarios:

- I went for lunch with some interesting person - let's call him Bob - and after the meeting, I've noted Bob's phone number on the piece of paper. Then I've met with you and told you how nice it was to meet Bob. You asked me for his phone number. You noted it down on your own piece of paper. Bob's phone number is a **value type**. You created a **copy** of the actual piece of information. Now, if I **change** what I wrote down on my piece of paper, it **will not affect** what you have on yours. The variable - in this case, the piece of paper - is a piece of information itself.
- I went to the library and asked where I can find Jon Skeet's "C# in depth". The librarian gave me a piece of paper with the number of the row and the bookshelf. I went over there and enjoyed reading. Then I met with you and

told you how great this book is. You've noted down the "**address**" of the book. In this metaphor, the book is a reference type. I can only give you the address - called "**reference**" in C#. If you go to the library, and you draw some doodles on the pages (please don't!) it will also affect the book I would be reading - because this is the same book! There is only one copy in the memory, but it can be pointed to by multiple references.

Now, let's look closer and more technically at the differences between value and reference types.

The first basic difference is that all reference types inherit from System.Object whereas all value types inherit from System.ValueType.

Another difference is that when a **value type** is passed as a **parameter**, its **copy** is given to the method. When a **reference type** is passed as a parameter, a **copy of the reference** is given to the method. Let's see some code to understand what it means:

```
private static void AddOne(int number)
{
    ++number;
}

private static void AddOneToList(List<int> list)
{
    list.Add(1);
}
```

As you can see we defined two methods. Both of them alter the parameter that was passed to them. The fundamental difference is that the AddOne method takes a value type, while the AddOneToList takes a reference type. What do you think this code will print?

```
int a = 5;
Console.WriteLine($"Number is {a}");
AddOne(a);
Console.WriteLine($"Now number is {a}");
```

Well, it will print "5" twice! This is because the a variable has been passed to the AddMethod by a copy. The AddMethod incremented the copy, so the original a variable has not been affected.

```
Number is 5
Now number is 5
```

Let's see a similar situation, but for the reference types:

```
var list = new List<int>();
Console.WriteLine($"List contains {list.Count} elements");
AddOneToList(list);
Console.WriteLine($"Now list contains {list.Count} elements\n");
```

What do you think will be printed? Well, since the List is a reference type, the AddOneToList does not operate on its copy, but on the same object that is referenced by the list variable. Because of that, the number of elements in the list variable has been incremented by one.

```
List contains 0 elements
Now list contains 1 elements
```

Another difference between value and reference types is that when assigning the value of the value type to the new variable, **a copy is created**. The change in the original object will not affect the new object. For reference types, when assigning the value to the new variable, **only the reference is copied**, while the original object still exists in one copy only. That means that the change in the original object will also affect what is stored in the new variable.

Let's see this in code for value types:

```
int b = 10;
int c = b;
++c;
Console.WriteLine($"Number 'b' is {b}");
Console.WriteLine($"Number 'c' is {c}\n");
```

This is the result printed to the console:

```
Number 'b' is 10
Number 'c' is 11
```

Incrementing the c variable did not affect the b variable, since a copy was created at the assignment.

Now, let's take a look at the reference types:

```
List<int> listB = new List<int> { 1, 2, 3 };
List<int> listC = listB;
listC.Add(4);
Console.WriteLine($"listB contains {listB.Count} elements");
Console.WriteLine($"listC contains {listC.Count} elements\n");
```

For this code, the result will be:

```
listB contains 4 elements
listC contains 4 elements
```

At assignment, no copy of the listB was created. Only the reference was copied, and it still points to the same object. Because of that adding an element to the listC variable also affected listB - because listC and listB point to the same object.

There are also some more technical differences between value types and reference types. Firstly, all value types are sealed, which means other types can't inherit from them. Because of that, value types can't have virtual or abstract members. See the "What is the purpose of the "sealed" modifier?" lecture for more information.

Secondly, value types are stored on the **stack** whereas reference types are stored on the **heap** (only the reference itself is stored on the stack). The value of the value type variable is cleaned out from the stack when the code execution leaves the scope this variable lived in. For reference types it is not the case - the object addressed by the reference will be cleaned up by the Garbage Collector and the exact time of that is unknown.

Let's summarize the differences between value types and reference types:

- All reference types inherit from System.Object whereas all value types inherit from System.ValueType.
- When a value type is passed as a parameter, its copy is given to the method. When a reference type is passed as a parameter, a copy of the reference is given to the method.
- When assigning the value of the value type to the new variable, a copy is created. The change in the original object will not affect the new object. For

reference types, when assigning the value to the new variable, only the reference is copied, while the original object still exists in one copy only. That means that the change in the original object will also affect what is stored in the new variable.
- All value types are sealed, which means other types can't inherit from them. Because of that, value types can't have virtual or abstract members.
- Value types are stored on the stack whereas reference types are stored on the heap (only the reference itself is stored on the stack).
- The value of the value type variable is cleaned out from the stack when the code execution leaves the scope this variable lived in. For reference types it is not the case - the object addressed by the reference will be cleaned up by the Garbage Collector and the exact time of that is unknown.

Tip: other interview questions on this topic.

- "**What will happen if you pass an integer to a method and you increase it by one in the method's body? Will the variable you passed to the method be incremented?**"
  The number will be increased in the scope of the method's body, but the variable outside this method will stay unmodified because a copy was passed to the method.

- "**Assuming you want the modification to the integer parameter to affect the variable that was passed to a method, how would you achieve that?**"
  By using ref parameter. See the question "What is the difference between the "ref" and the "out" keywords?".

5. What is boxing and unboxing?

Brief summary: Boxing is the process of wrapping a value type into an instance of a type System.Object. Unboxing is the opposite - the process of converting the boxed value back to a value type.

To understand what boxing and unboxing are, it is essential to understand what value and reference types are. You can learn about them from the "What is the difference between value types and reference types?" lecture.

Boxing is the process of converting a **value type to the System.Object type**. **Unboxing** is the **opposite** - the process of converting the boxed value back to value type. When the value is boxed, it is wrapped inside an instance of the System.Object class and stored on the heap.

As we know, value types are stored on the **stack** while reference types are stored on the **heap**. Only the reference itself (so an "address" or "pointer" to the object stored on the heap) is stored on the stack. Let's see a short piece of code:

```
int number = 5;
string word = "abc";
```

In this situation the following data is stored in the memory:

- On the stack
    - The value of number variable (5), as an integer is a **value** type
    - The reference to the word variable stored on the heap, as string is a **reference** type
- On the heap
    - The value of the word variable ("abc")

Let's box the value of the number variable:

```
int number = 5;
object boxedNumber = number;
```

A new variable of type object is created. Object is a reference type, so its value is stored on the heap. Let's see the state of the stack and the heap now:

- On the stack
    - The value of number variable (5), as an integer is a value type
    - The reference to the word variable stored on the heap
    - The reference to the boxedNumber variable stored on the heap
- On the heap
    - The value of the word variable ("abc")
    - The value of the boxedNumber variable (5)

As you can see, boxing is done implicitly. On the other hand, the unboxing must be done explicitly by using a cast:

```
int number = 5;
object boxedNumber = number;
int unboxedNumber = (int)boxedNumber;
```

Unboxing unwraps the original value from the object and assigns it to a value type variable.

The unboxing requires the exact type match. For example, this would throw an exception, because integer is not the same as short:

```
//this will throw
//int unboxedShortNumber = (int)boxedShortNumber;

short otherShortNumber = 3;

//this will work fine - no boxing or unboxing here
int otherShortNumberCastToInt = (int)otherShortNumber;
```

Without boxing and unboxing, casting a short to an integer works fine.

Please be aware that boxing and unboxing come with a **performance penalty**. Unlike regular variables assignment, boxing requires the creation of a new object and allocating memory on the heap for it. The unboxing requires a cast, which is also computationally expensive.

Now we know how boxing and unboxing are done and what exactly they do. **But what's their use?** Well, boxing and unboxing are necessary for providing a **unified type system** - that is, that we can treat any variable in C# as an **object**. Without boxing and unboxing, we couldn't have the ultimately generic code that accepts

any type of variable - we would have to distinguish value and reference types and possibly provide separate implementations for both of them. That was particularly useful before the generic types were introduced, and classes like ArrayList (used for storing any type of data) were commonly used. Even nowadays it is still used, for example in ADO.NET which is used to store objects in databases - at some point, this framework treats every piece of data as an object. Without boxing, it wouldn't be able to handle value types.

Tip: other interview questions on this topic:

- "**What is the penalty for using boxing and unboxing?**"
  The main penalty is performance - when boxing, a new object must be created, which involves allocating memory for it. The unboxing requires a cast which is also expensive from the performance point of view.

- "**Is assigning a string to a variable of type object boxing?**"
  No, because string is not a value type. The point of boxing is to wrap a value type into an object (which is a reference type).

6. What are the three main types of errors?

Brief summary: The three main types of errors are:
1. compilation errors, also known as syntax errors, reported by the compiler
2. runtime errors, thrown during program execution
3. logical errors, occurring when the program works without crashing but it does not produce a correct result

There are three main types of errors in programming:
- Compilation errors (also known as syntax errors)
- Runtime errors
- Logical errors

Compilation errors, also known as syntax errors, occur when the programmer makes a syntax mistake during writing the code. They are detected by the compiler and shown to the programmer via the IDE (Visual Studio, Rider, or any other). Let's see a simple compilation error caused by a missing semicolon:

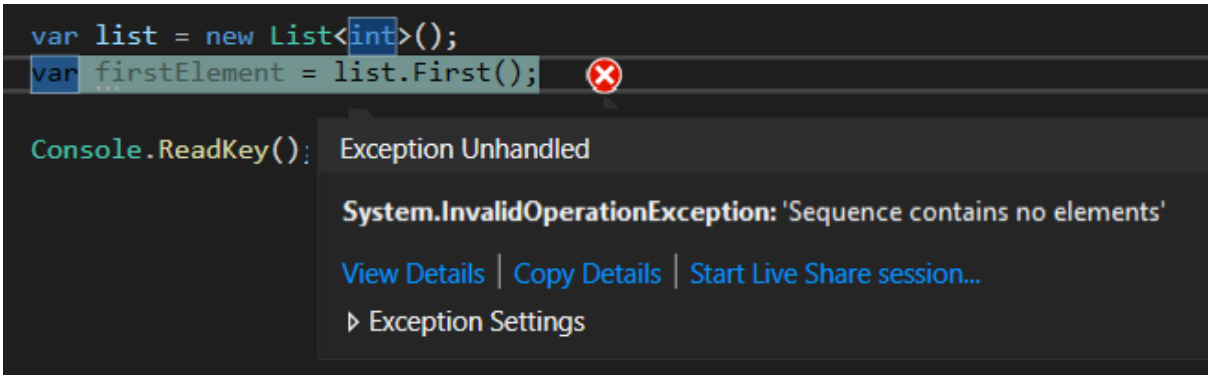
```
var number = 5
```

In this case, the error is shown in the Visual Studio:

	Code	Description	Project	File	Line	Suppression State
⊗	CS1002	; expected	TypesOfErrors	Program.cs	9	Active

Compilation errors are probably most common, and also easiest to fix. A tip on how to fix them is usually shown in the IDE. Unless all compilation errors are fixed, the program will not be compiled, and of course, it will not be run.

Runtime errors are the ones that occur when the program is running. The program must have been compiled correctly (so no compilation errors were present), but something goes wrong during the program execution. For example, we may try to access the first element of an empty list:



It may be a bit more tricky to fix a runtime error - it's best to use the debugger to stop the program right before the error occurs, to see the exact state of the application.

Logical errors happen when the program is running without crashing, but it does not produce a correct result.

```
    var sentence = MergeWords("A", "little", "duck", "swims", "in", "a", "pond");
    Console.WriteLine(sentence);

    Console.ReadKey();
}

private static object MergeWords(params string[] words)
{
    return string.Join("", words);
}
```

The above code was supposed to merge words into a sentence. Unfortunately, the result does not look good:

```
Alittleduckswimsinapond
```

The programmer made a mistake when using the Join method from the string class - the separator should be a space, not an empty string.

The best way of making sure logical errors are not present in our application is to create a solid suite of unit tests.

Tip: other interview questions on this topic:

- "**What type of errors do unit tests protect us from?**"
  Both runtime errors and logical errors.

- "**What's the C#'s mechanism for handling runtime errors?**"
  Exceptions - they are used for handling runtime errors. See the lecture 7 "How are exceptions handled in C#?" for more information.

7. How are exceptions handled in C#?

Brief summary: Exceptions are handled by try-catch-finally blocks. Try contains code that may throw exceptions, catch defines what should be done if an exception of a given type is thrown, and finally is executed no matter if the exception was thrown or not.

Every program may encounter some exceptional situations that need to be somehow handled. Let's take a look at some of them:

- **System errors**, like running out of memory - OutOfMemoryException is thrown
- **Numeric errors**, like when trying to increment an int that already has the maximum int value assigned - OverflowException is thrown
- **Parsing errors**, like trying to parse "hello" to an integer - FormatException is thrown
- **Operation errors**, like trying to remove the first element from an empty list - ArgumentOutOfRangeException is thrown

Remember that in C# every exception type derives from class **Exception**.

As developers, we are responsible for ensuring that all exceptions are properly handled in our application. Imagine that you want to use a calculator on your phone, and the application crashes every time you do something "illegal" like dividing by zero. It would be pretty annoying. It is much better to handle such a situation, by for example presenting the user some kind of error message that explains what is wrong.

In C# we use try-catch-finally blocks to handle exceptions:

```
private static int? DivideNumbers(int a, int b)
{
    try
    {
        return a / b;
    }
    catch (DivideByZeroException ex)
    {
        Console.WriteLine($"Attempt to divide by zero." +
            $" Exception message: {ex.Message}");
        return null;
    }
    finally
    {
        Console.WriteLine("Executing finally block");
    }
}
```

- In the **try** block, we put the code that may throw an exception
- In the **catch** block, we define what kind of exception we want to handle. If not specified, any kind of exception will be caught
- In the **finally** block we put code that needs to be executed no matter if the operation was successful or not

Let's take a closer look at each of those blocks:

- **Try block**
    - Any code that we expect to possibly throw an exception should be put in the try block. But we can't expect everything. Exceptions are often, well, unexpected. For example, it's usually hard to predict OutOfMemoryException - it can happen almost anywhere. Or, for example, NullReferenceExceptions - it wouldn't make much sense to wrap every piece of code that uses objects that may be null in the try-catch blocks - that would cause terrible noise in the code.
    - The solution is to only use local try blocks when we truly expect some error-causing situations, and we can handle them in some specific way. All other cases should be handled by a global try-catch block. It is a try-catch block that wraps the entry point to the application (for example in the Main function in console applications), thus catching

all exceptions that hadn't been caught in a more specific context. The downside of such a global try block is that it cannot be very specific - since it catches any type of an exception. Usually, such exceptions are only logged and presented to the user in some kind of error window.

```
class Program
{
    static void Main(string[] args)
    {
        //global try-catch block
        try
        {
            Application.Run();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Exception was thrown by the application," +
                $" error message is: {ex.Message}");
        }
    }
```

- **Catch block**
  - Catch block defines what kind of exception will be handled, and how it will be done:

```
catch (DivideByZeroException ex)
{
    Console.WriteLine($"Attempt to divide by zero." +
        $" Exception message: {ex.Message}");
    return null;
}
```

  - We can omit the exception variable name if we do not intend to use it:

```
catch (DivideByZeroException)
{
    Console.WriteLine($"Attempt to divide by zero.");
    return null;
}
```

  - We can also omit the exception type if we don't care about it and want to catch any type of exception:

```
catch
{
    Console.WriteLine($"Attempt to divide by zero.");
    return null;
}
```

- We can have multiple catch blocks that handle different types of exceptions:

```
private static int? ElementAtIndex(int[] numbers, int index)
{
    try
    {
        return numbers[index];
    }
    catch (NullReferenceException)
    {
        Console.WriteLine($"Input array is null");
        return null;
    }
    catch (IndexOutOfRangeException)
    {
        Console.WriteLine($"Index {index} does not exist in input array");
        return null;
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Unexpected error, see message: {ex.Message}");
        return null;
    }
}
```

- Please note that in the case of multiple catch blocks, when the exception is thrown it will be caught by the first catch block that handles the matching exception type. Because of that, we should always write catch blocks from the most specific to the most generic. Remember that the most generic type of exception is the Exception class, a base class for all other exceptions. Let's change the above code so the most generic exception is at the top:

```
private static int? ElementAtIndex(int[] numbers, int index)
{
    try
    {
        return numbers[index];
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Unexpected error, see message: {ex.Message}");
        return null;
    }
    catch (NullReferenceException)
    {
        Console.WriteLine($"I
        return null;
    }
    catch (IndexOutOfRangeException)
    {
        Console.WriteLine($"Index {index} does not exist in input array");
        return null;
    }
}
```

```
class System.NullReferenceException
The exception that is thrown when there is an attempt to dereference a null object reference.

CS0160: A previous catch clause already catches all exceptions of this or of a super type ('Exception')
```

- As you can see, it does not compile. C# compiler is smart enough to know such code is incorrect. If it did compile, and if the "numbers" parameter would be null, such exception would be handled by the first, generic catch block. It is not what we want - we want to have specific behavior for NullReferenceException and for IndexOutOfRangeException. Always put more specific exceptions at the top, and more generic at the bottom of the catch blocks list.

- **Finally block**
  - The code in the finally block will always be executed, no matter if the exception was thrown in the try block or not. It is executed last, after the try block and any catch block, if an exception was thrown. It is used to clean up some resources that were used in the try block. For example, if you opened a database connection in the try block, you should close it in the finally block - to make sure it will be closed no matter if an exception was thrown.
  - You might think "Why close the database connection in the finally block if I can do it in the finalizer?". Technically, yes, but remember that we do not know when Garbage Collector will be fired and when it will execute finalizers. By cleaning resources in the finally block, we are ensuring it is done as soon as possible. It is especially important when the resources are limited - for example, a database may not allow more than one connection to be opened, and then it would be blocked until Garbage Collector would close the connection.

Tip: other interview questions on this topic:

- "**Is it possible to have multiple catch blocks after a try block?**"
  Yes. You can catch any number of exceptions. It is important to first catch the more specific, and then more generic exceptions.

- "**How to ensure some piece of code will be called, even if an exception was thrown?**"
  You should put this code in the finally block.

- "**What is the base type for all exceptions in C#?**"
  System.Exception.

8. What are the types of access modifiers in C#?

Brief summary: public, internal, protected, protected internal, private protected and private

Note: for the sake of brevity we will only consider classes here, but access modifiers work the same way for classes, structs, and records. Please note that not all access modifiers are supported by structs (as they do not support inheritance, using access modifiers like protected on struct's members would not make sense).

There are **six** access modifiers in C#. The order is from least to most restrictive:

- **Public** - the type or member can be used by any other type from any assembly.
- **Internal** - the type or member can be used by any other type from the same assembly it is defined in. It can't be used outside this assembly.
- **Protected** - the type or member can be used only in the same class, or in the class that inherits from this class (no matter in what assembly)
- **Protected internal** - within the same assembly it works as internal - so it can be used by all other types. Outside this assembly it works as protected - it can be used only by types inheriting from this class.
- **Private protected** - within the same assembly it works as protected - the type or member can be used only in the declaring class or in the classes that inherit from it. Outside this assembly, it can't be accessed, even by classes inheriting from this class.
- **Private** - the type or member can be used only by the code in the same class.

Tip: other interview questions on this topic:

- "**What is the difference between protected and private protected access modifiers?**"
  The difference is that the type defined with private protected access modifier is only available in the declaring assembly. Except for that, both work similarly: the types or members can be accessed by the class they are defined in or in the derived classes.

- "**What is the difference between private and private protected access modifiers?**"
  The type or member defined with the private access modifier can only be accessed in the containing class. If private protected access modifier would be used instead, the type or member could be also used by derived classes within the same assembly.

- "**What is the difference between protected and protected internal access modifiers?**"
  Types or members that are protected internal work as they are both protected and internal at the same time - so within their containing assembly they are accessible by any class (that's how internal works) and outside this assembly, they are available only in derived classes (that's how protected works).

9. What are the default access modifiers in C#?

Brief summary: The default access modifier at the namespace level is internal. At the class level, it is private.

First, let's understand what a "default access modifier" means. It is an access modifier that will be applied to a type or member in case the programmer doesn't declare any access modifier explicitly.

The shortest answer to the question "What are the default access modifiers in C#?" is "**They are the most restrictive access modifiers that are valid in the given context**".

Let's start with a short reminder on access modifiers in C#:

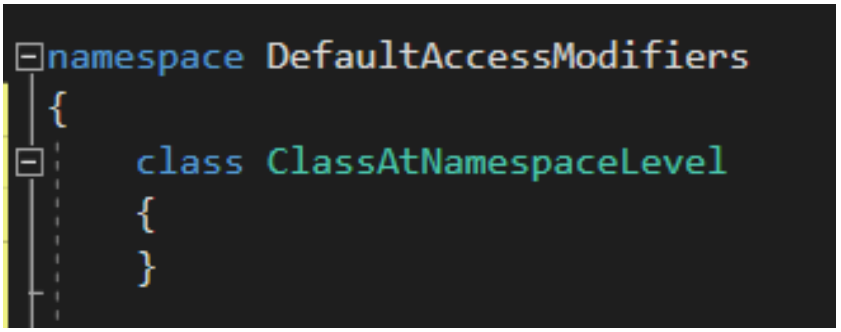
- **Public** - the type or member can be used by any other type from any assembly.
- **Internal** - the type or member can be used by any other type from the same assembly it is defined in. It can't be used outside this assembly.
- **Protected** - the type or member can be used only in the same class, or in the class that inherits from this class (no matter in what assembly)
- **Protected internal** - within the same assembly it works as internal - so it can be used by all other types. Outside this assembly it works as protected - it can be used only by types inheriting from this class.
- **Private protected** - within the same assembly it works as protected - the type or member can be used only in the declaring class or in the classes that inherit from it. Outside this assembly it can't be accessed, even by classes inheriting from this class.
- **Private** - the type or member can be used only by the code in the same class or struct.

For more information on access modifiers in C# please see the "What are the types of access modifiers in C#?" lecture.

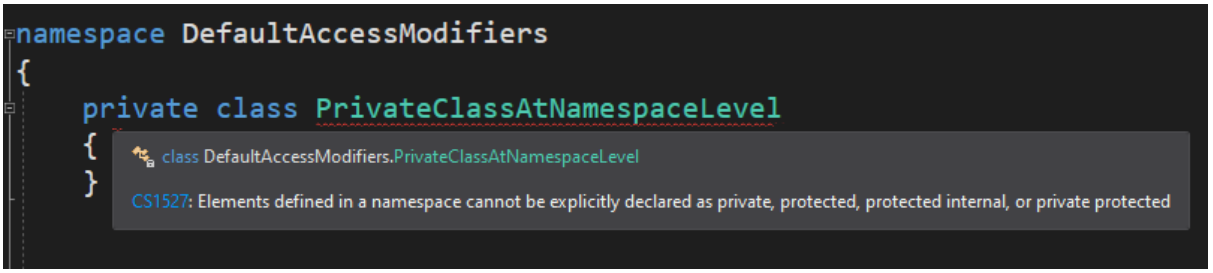
In C# there are two levels at which we can declare types or members:

- **Namespace level** - where we declare classes, structs, records, enums, delegates, and interfaces
- **Class/struct/record level** - where we declare fields, properties, methods, events, as well as inner classes, structs, records, enums, delegates, and interfaces

Let's first consider the **namespace level**. Let's have a class without any access modifier declared at namespace level:



What are the access modifiers we could use with this class? It can only be internal or public. The other access modifiers do not make any sense at the namespace level, and the compiler tells us this:



As we said before - the default access modifier is the most restrictive access modifier that is valid in the given context. In this case, internal is more restrictive than public. **That means at the namespace level the default access modifier is internal.**

Within a class it is simple - all members of a class can be declared private, and private is the most restrictive access modifier of all. **That means, the default access modifier at the class level is private**. Please note that nested classes will also be private, unlike classes declared at the namespace level.

```
//this class is internal by default
class ClassAtNamespaceLevel
{
    //this is private by default
    int number;

    //all access modifiers other than private are non-default
    public int publicNumber;


    //this class is private by default,
    //because it is declared at class level
    class InnerClass
    {

    }
}
```

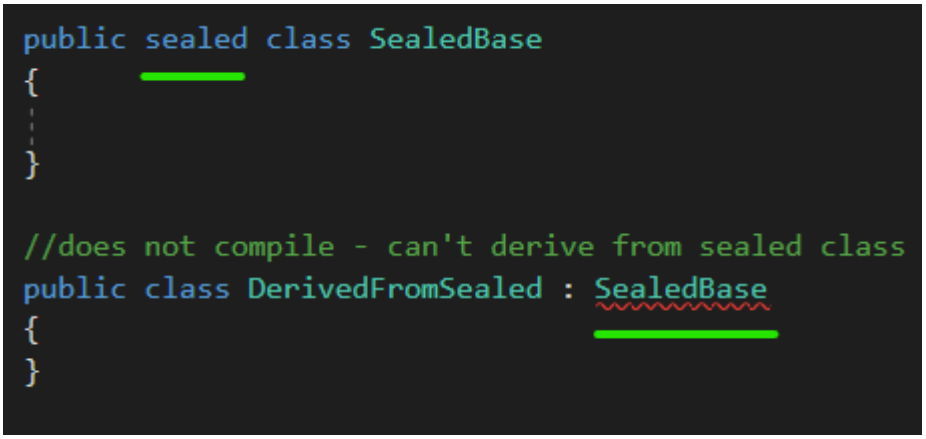
Tip: other interview questions on this topic:

- "**What's the default access modifier for a class?**"
  Internal, unless the class is not nested. If it is, the answer is "private".

10. What is the purpose of the "sealed" modifier?

Brief summary: The sealed modifier prevents a class from being inherited, or an overridden method from further overriding.

The sealed modifier **prevents a class from being inherited** from:



We can also use it with an **overridden** method, to **prevent further overriding** in child classes:

```
public class Base
{
    public virtual void DoSomething()
    {
        Console.WriteLine("Base class");
    }
}

public class Derived : Base
{
    public override sealed void DoSomething() //we are sealing this method
    {
        Console.WriteLine("Derived class");
    }
}

public class DerivedFromDerived : Derived
{
    //does not compile - this method was sealed in Derived class
    public override void DoSomething()
    {
    }
}
```

Only methods with the "override" modifier can be sealed.

What might be the **use cases** for using the sealed modifier?

- It can be applied when a developer expects that overriding some functionality might make it no longer work.
- It can be applied when a developer doesn't expect any reasonable need for his class to be overridden. For example System.String class is sealed, as providing custom behavior of strings probably would not make much sense. On the other hand, I would recommend being **careful** with such an assumption - other developers might have some interesting ideas on how to implement custom behavior in the inheriting classes, and they would not like the "sealed" modifier preventing them from doing so.
- Another use case could be using the "sealed" modifier with a class implementing some security features - we don't want anyone to be able to override those features in child classes

- Sealing a class might increase the overall performance of the application because it tells the CLR (Common Language Runtime) that it doesn't need to look for an overridden method further down in the hierarchy.

Please be aware that there are some **disadvantages** of using the sealed modifier:

- Sealed modifier by design is preventing the inheritance from the class or overriding a method - it might not be desirable for many developers, as they might have valid reasons to do so. It's very annoying to want to provide a custom implementation of some logic, and being denied to do so because of the "sealed" modifier.
- "Sealed" modifier prevents the ability for the class to be mocked in tests. Mocking is a topic beyond the junior level, but in short, it's about creating a special "stub" of a class used for testing purposes - for example, a class that pretends to be accessing the database, but actually it just returns some fixed values. In most frameworks mocking is based on deriving from the class that is supposed to be mocked. Mocking is critical for creating unit tests, so please be considerate of it when deciding to seal a class or a method.

Let's summarize:

- You can add the sealed modifier to a class to prevent it from being inherited. It cannot be added to an abstract class, because abstract classes by definition exist only to be inherited from.
- You can add the sealed modifier to an overridden method, to prevent further overriding in child classes. Only methods with the "override" modifier can be sealed.

Tip: other interview questions on this topic:

- "**How would you prevent the class from being inherited?**"
  By making it sealed.

- "**How can you make the class inheritable, but prevent specific methods from being further overridden?**"
  By making the overridden method sealed.

- "**Can you make an abstract class sealed?**"
  No. The whole point of an abstract class is to inherit from it. Making it sealed makes no sense.

11. What is the purpose of the "params" keyword?

Brief summary: The “params” keyword allows us to pass any number of parameters of the same type to a method.

Imagine you want to create a function that sums up any number of provided integers. You could have an array of integers as input parameters:

```
static int Sum(int[] numbers)
{
    var sum = 0;
    foreach (var number in numbers)
    {
        sum += number;
    }
    return sum;
}
```

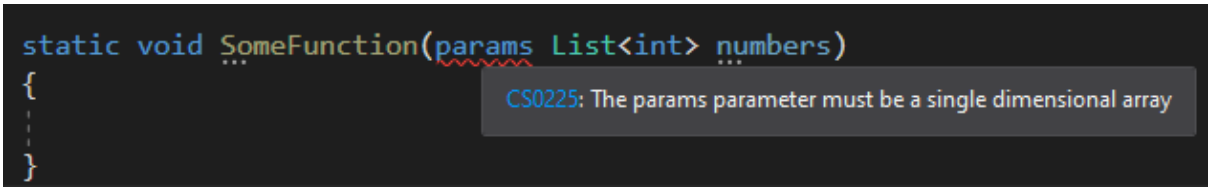
And then, use it like this:

```
Sum(new[] { 1, 2 });
Sum(new[] { 1, 2, 3, 4 });
Sum(new int[] { });
```

It is a bit clumsy, though. Declaring an array every time we want to call the Sum function makes the code longer and less readable. Instead, we could use the **params** keyword with the input array parameter.

```
static int Sum(params int[] numbers)
{
    var sum = 0;
    foreach (var number in numbers)
    {
        sum += number;
    }
    return sum;
}
```

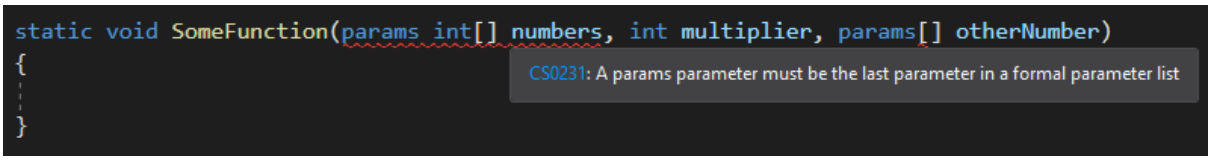
Params keyword specifies a method parameter that takes any number of arguments. The parameter type **must be a single-dimensional array** - otherwise, we will get a compiler error.



So now, we can call the Sum method like this:

```
Sum(1, 2);
Sum(1, 2, 3, 4);
Sum();
```

This code is much more readable and concise. Please note that the parameter with **params keyword must be the last** in the parameters list. It makes sense - since there can be any number of values there, the compiler would not know if the following parameter is still the member of the params array or if it is the next parameter in the parameters list.



Calling this method like this would be confusing:
SomeFunction(1,2,3,4,5);
How could .NET know which number belongs to numbers, and which to otherNumbers parameter?

Tip: other interview questions on this topic:

- "**Why must the parameter with params modifier be the last in the parameters list?**"
  Because if it wasn't, and there was another parameter after it, the compiler would not know if this parameter belongs to the params array or not.

12. What is the difference between a class and a struct?

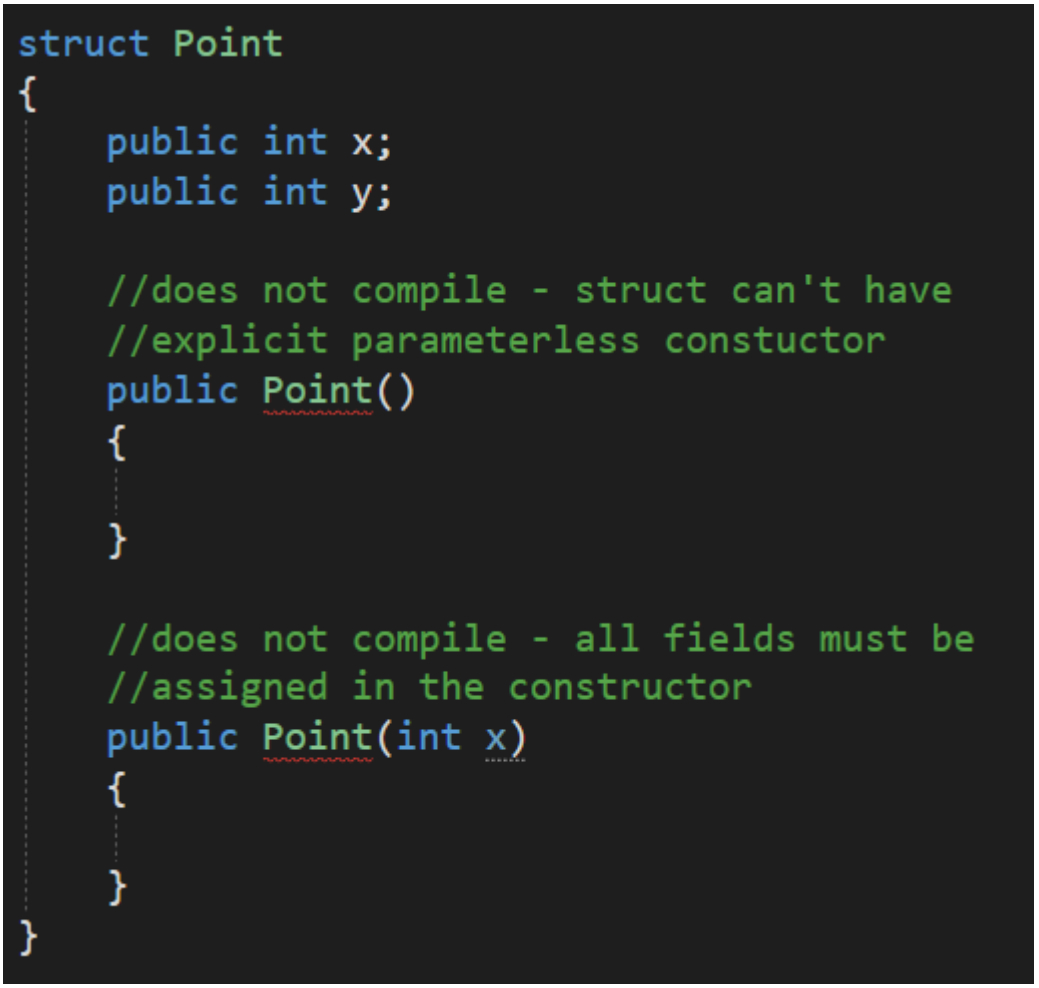
Brief summary:

1. Structs are value types and classes are reference types.
2. Structs can only have a constructor with parameters, and all the struct's fields must be assigned in this constructor.
3. Structs can't have explicit parameterless constructors.
4. Structs can't have destructors.

There are several differences between structs and classes:

- **Structs are value types and classes are reference types.** Please refer to the "What is the difference between value types and reference types?" lecture for more information. Structs, like all value types:
    - inherit from System.ValueType
    - are passed by copy
    - are sealed (which means, they cannot be inherited)
    - are stored on the stack

- Structs can only have a **constructor with parameters**, and all the struct's fields must be assigned in this constructor. Structs can't have explicit parameterless constructors. (The reason for that is quite low-level and technical, and probably beyond Junior's level of knowledge. If you are curious, I recommend this thread, where Jon Skeet explains this topic thoroughly: https://stackoverflow.com/questions/333829/why-cant-i-define-a-default-constructor-for-a-struct-in-net)



```
struct Point
{
    public int x;
    public int y;

    //does not compile - struct can't have
    //explicit parameterless constuctor
    public Point()
    {

    }

    //does not compile - all fields must be
    //assigned in the constructor
    public Point(int x)
    {

    }
}
```

- **Structs can't have destructors** (finalizers). A destructor is a special method that is executed when the object is being removed from the memory. To understand why structs can't have destructors, you must remember that they are value types, and whenever we assign one struct variable to another or pass a struct as a parameter, a copy of this struct is created. Now, imagine you would have a struct that manages a database connection, and in the Finalize method, you would like to close this connection. Then, let's say you pass your struct as a parameter to a method. A copy of the struct is created, and at the end of the method's scope, the struct's Finalize would be called, closing the database connection. But what about the struct you actually passed to a method? The one that a copy was created from? It would suddenly stop working, as the connection would be closed. This is why a struct can't have a destructor defined.

```
//does not compile - structs can't have destructors
public ~Point()
{

}
```

When should we use structs?

- The type is logically small (for example it represents a single value, like int, double, or bool)
- It is small from the memory point of view (under 16 bytes)
- It is immutable (once an instance is created, it is not modified)
- It is commonly short-lived
- It is commonly embedded in other objects
- You want value type semantics (creating a copy on assignment, passing a parameter by copy)
- It will not be boxed frequently

If some of those criteria are not met, you should rather use a class.

Tip: other interview questions on this topic:

- "**What is the base type for structs?**"
  System.ValueType.

- "**Is it possible to inherit from a struct?**"
  No, all structs are sealed.

- "**How would you represent a point in the cartesian coordinate system?**"
  I would create a struct that has two float readonly properties - X and Y.

13. What are partial classes?

Brief summary: Partial classes are classes that are split over two or more source files. All parts are combined when the application is compiled. It is also possible to declare partial structs, interfaces, and methods.

Partial classes are classes that are **split over two or more source files**. All parts are combined when the application is compiled. It is also possible to declare partial structs, interfaces, and methods.

Let's see a simple example. This class is divided into two parts, one in file DuckPartOne…

```
partial class Duck
{
    private void Quack()
    {
        Console.WriteLine("Quack, quack, I'm a duck");
    }
}
```

…and other in DuckPartTwo:

```
partial class Duck
{
    private void Swim()
    {
        Console.WriteLine("Swimming in a pond");
    }
}
```

There is no difference between the above code split into two parts and the code below:

```
class Duck
{
    private void Quack()
    {
        Console.WriteLine("Quack, quack, I'm a duck");
    }

    private void Swim()
    {
        Console.WriteLine("Swimming in a pond");
    }
}
```

As you can see, the **partial** keyword is used to create a partial class.

Partial interfaces and structs work exactly the same as partial classes.

The partial methods are slightly different - we declare the signature of the method in one file…

```
partial class Duck
{
    private void Quack()
    {
        Console.WriteLine("Quack, quack, I'm a duck");
    }

    public partial void Fly();
}
```

..and the body in another:

```
partial class Duck
{
    private void Swim()
    {
        Console.WriteLine("Swimming in a pond");
    }

    public partial void Fly()
    {
        Console.WriteLine("Flying high in the sky");
    }
}
```

The partial classes are not commonly used, but there are some situations when they might be **useful**:

- When the class is large, we may consider splitting it into smaller files. It is particularly useful when many developers work on the same class at the same time. It makes merging the work easier. On the other hand, if a class is so large that it's a good idea to split it, it is even a better idea to refactor it.
- When some of the code is automatically generated:
    - For example, when working on Windows Forms applications some of the code is generated when we create the form with the graphical tools. The developer uses the graphical interface to create, let's say, a button, and the code generator creates an event handler to handle clicking on the button
    - A similar case is the code generated by Entity Framework (it's a tool used to allow storing C#'s objects as database entities easily)

  If partial classes didn't exist, those code-generating tools would constantly interfere with the code we write by hand.

Tip: other interview questions on this topic:

- **"What's the use of partial classes?"**
  Splitting a large class into more pieces, which can be useful when many programmers contribute to the development of this class - makes the source control easier. Also, partial classes are useful when some code is automatically generated - we can have the generated code in one part, and the human-written code in another.

- "**Is it possible to create a partial struct?**"
  Yes. We can create partial classes, structs, and interfaces, as well as partial methods.

14. What does the "new" keyword do?

Brief summary: The "new" keyword is used in three different contexts:

1. The new operator, which creates a new instance of a type
2. The new modifier, which is used to explicitly hide a member method from a base class in the derived class
3. The new constraint, which specifies that a type argument in a generic class must have a parameterless constructor

In C#, the "new" keyword can be used in **three** different contexts:

- **new operator**, which creates a new instance of a type:

  We simply use it to invoke the constructor from a class to create a new object of a type:

```
var person = new Person("Rachel", 34);
```

  Please note that since C# 9 we can omit the type name during object creation if the type is known:

```
Person person = new("Rachel", 34);
```

  We can also create an object with the object initializer:

```
var person = new Person { Name = "Steve", Age = 45 };
```

  …or we can create a collection with the collection initializer:

```
var currencies = new Dictionary<string, string>
{
    ["USA"] = "USD",
    ["Great Britain"] = "GBP"
};
```

We can create an array:

```
var numbers = new int[4];
```

Or instantiate anonymous type:

```
var person = new { Name = "Anna", Age = 55 };
```

- **new modifier**, which is used to explicitly hide a member method from a base class in the derived class. See the "What is the difference between method overriding and method hiding?" lecture for more information.

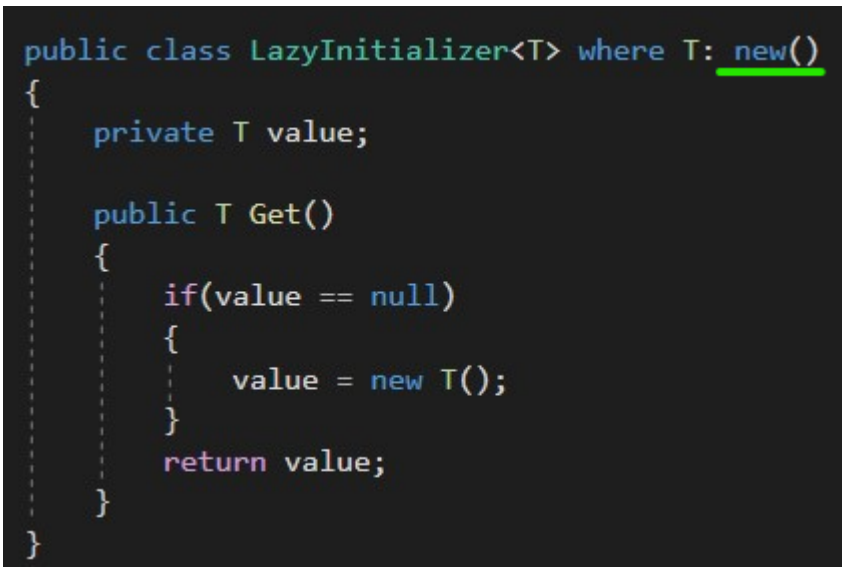
- **new constraint**, which specifies that a type argument in a generic class must have a parameterless constructor:

```
public class LazyInitializer<T>
{
    private T value;

    public T Get()
    {
        if(value == null)
        {
            value = new T();
        }
        return value;
    }
}
```

T in LazyInitializer<T>

CS0304: Cannot create an instance of the variable type 'T' because it does not have the new() constraint

To fix this error, we must add the "new" constraint:



```
public class LazyInitializer<T> where T: new()
{
    private T value;

    public T Get()
    {
        if(value == null)
        {
            value = new T();
        }
        return value;
    }
}
```

Now, only types with parameterless constructors can be used as T.

Tip: other interview questions on this topic:

- **"How can you create a new object of type T in a generic class?"** *By adding the new constraint to the type T.*

- **"How do you hide a base class member method in the derived class?"** By using the new modifier.

- **"How many uses does the "new" keyword have?"** *Three - new operator, new modifier, and new constraint.*

15. What is the purpose of the "static" keyword?

Brief summary: The "static" keyword can be used in two contexts:

1. **static modifier** - used to define static classes, as well as static members in classes, structs, and records
2. **using static directive** - used to reference static members without needing to explicitly specify their name every time

The "static" keyword can be used in two contexts:

1. **static modifier**
   The static modifier is used to declare static members in classes, structs, and records. Static members belong to the type itself rather than to a specific object. Let's take a look at a couple of uses of the static modifier:

```
class Box
{
    public static int MaxCount = 50;

    private List<string> elements = new List<string>();

    public void Add(string element)
    {
        if(elements.Count < MaxCount)
        {
            elements.Add(element);
        }
    }
}
```

   The purpose of this class is to hold up to 50 strings. The class itself should not be static - because we want to be able to have multiple, independent instances of boxes - but the MaxCount property does not belong to any specific instance. It belongs to the class as a whole.

```
var box1 = new Box();
var box2 = new Box();
var invalidMaxCount = box1.MaxCount;
var maxCount = Box.MaxCount;
```

As you can see from the example above, we can't access static members from a specific instance. We must refer to the type name.

Note: all const fields in C# are implicitly static. This makes sense - since the value is constant, it will be the same for all instances of a class, so it can be shared between them and belong to the type itself. That's why in the Box class it would be reasonable to change the "static" modifier to the "const" modifier for the MaxCount field.

Another use of the static modifier is to declare a static method. If a method in a class does not reference any non-static members of this class, it's best to declare it as static:

```
class Box
{
    public static int MaxCount = 50;

    private List<string> elements = new List<string>();

    public void Add(string element)
    {
        if(elements.Count < MaxCount)
        {
            elements.Add(element);
        }
    }

    //can't be made static as it refers
    //to non-static elements field
    public int GetCurrentCount()
    {
        return elements.Count;
    }

    public static string FormatMaxCount()
    {
        return $"The max count for this Box is {MaxCount}";
    }
}
```

Similarly, if we want to call a static method, we do it on a type, not on a specific object:

```
var box1 = new Box();
var box2 = new Box();
var elementsCount = box2.GetCurrentCount();
var maxCountFormatted = Box.FormatMaxCount();
```

The static modifier can be applied to fields, properties, methods, constructors, events, and operators. Since C# 8 you can add the static modifier to a local function, and since C# 9 - to lambda expression or an anonymous method.

The static modifier is also used to declare static classes. Please note that you can't have static structs or records (this is because since static objects cannot be instantiated, the behavior of a static struct or static record would be exactly the same as for static class). More on static classes can be found in the "What is a static class?" lecture.

2. **using static directive**

Let's take a look at the following code:

```
using System;

namespace DotNet50JuniorInterviewQuestions
{
    static class Geometry
    {
        public static double SquareArea(double side)
        {
            return Math.Pow(side, 2);
        }

        public static double CircleArea(double radius)
        {
            return Math.PI * Math.Pow(radius, 2);
        }

        public static double EquilateralTriangleArea(double side)
        {
            return (Math.Pow(side, 2) * Math.Sqrt(3)) / 4d;
        }
    }
}
```

As you can see, we are intensely using the static class Math here. We could use **the using static directive** to reference static members without needing to explicitly specify the Math namespace every time.

```
using static System.Math;

namespace DotNet50JuniorInterviewQuestions
{
    static class Geometry
    {
        public static double SquareArea(double side)
        {
            return Pow(side, 2);
        }

        public static double CircleArea(double radius)
        {
            return PI * Pow(radius, 2);
        }

        public static double EquilateralTriangleArea(double side)
        {
            return (Pow(side, 2) * Sqrt(3)) / 4d;
        }
    }
}
```

Tip: other interview questions on this topic:

- "**Let's say you have a console application, and you use "Console.WriteLine" and "Console.ReadLine" all the time. What can you do to make this code more concise**?"
  We can use "using static System.Console" and then use only "ReadLine" and "WriteLine" throughout the code.

- "**Let's say you have a variable of class Person called "john". How would you access a static method X from the Person class?**"
  I would not use the "john" instance - I would just call the "Person.X()" since X is a static method.

- "**Are const fields static?**"
  Yes, they are implicitly static.

- "**Let's say you have a class that works as a collection of objects. It has an Add method that is used to add an object to this collection. How would you keep track of the count of ALL elements that have ever been added to any instance of this class?**"

We could introduce a static "Counter" field that is incremented each time the "Add" method is called.

16. What is a static class?

Brief summary: A static class is a class that cannot be instantiated and can only contain static methods. It can work as a container for methods that just operate on input parameters and do not have to get or set any internal instance fields.

A **static class** is a class that cannot be instantiated. Consider a Calculator class, that provides a set of methods executing simple mathematical operations:

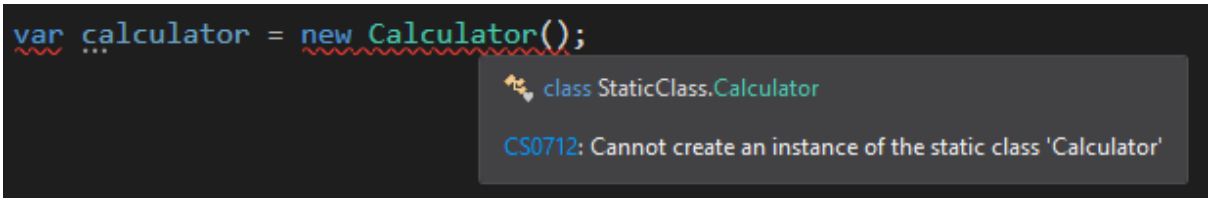
```
static class Calculator
{
    public static double Add(double a, double b)
    {
        return a + b;
    }

    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    public static double Power(double a)
    {
        return a * a;
    }
}
```

There's no point in having multiple instances of the Calculator class - they would all do exactly the same thing. They don't have any state that can make one instance different from another (unlike, for example, a List, that holds elements inside. One instance of a List can have different elements than another instance).

Since this class doesn't have any instance-specific state, it makes sense to make it static. This way, if we want to use methods from the Calculator class, we can call them directly on the Calculator type name:

```
var sum = Calculator.Add(4, 5);
```

If we tried to create an instance of the Calculator class, we would get a compiler error:



In general, **static classes are most often used as containers of static methods**.

There are plenty of built-in static classes that are commonly used:

- Console - providing methods like WriteLine, ReadLine, ReadKey - there wouldn't really make sense to have multiple Console instances, since the only job of this class is to communicate with the console window
- Math - similarly to our Calculator, it provides a set of mathematical operations and constants
- Environment - this class provides information about the environment the program works in, like the version of the operating system, current working directory, or the name of the user

As a rule of thumb, consider making a class static if it doesn't store any data that would make its instance unique - in other words, if all its methods can be made static.

There are a couple of rules regarding static classes:

- They can only have static members
- They cannot be instantiated
- They are sealed (which means, they cannot be inherited)
- Like all classes, they are derived from the System.Object class. They can't derive from any other class
- They cannot have instance constructors (the constructors that are used to create an instance)
- You can't use the "this" keyword inside a static class, as "this" refers to "current instance" and static classes cannot be instantiated

Static classes (as well as non-static classes) can contain **static constructors**. A static constructor allows us to initialize static fields. It is called when the static class is first used:

```
static class SystemMonitor
{
    private static readonly DateTime _startTime;

    static SystemMonitor()
    {
        _startTime = DateTime.UtcNow;
    }

    public static string Report()
    {
        return $"Monitoring the system for " +
            $"{(DateTime.UtcNow - _startTime).TotalSeconds} seconds";
    }
}
```

A static constructor must meet the following requirements:

- It must be parameterless
- It cannot have access modifiers
- Cannot be inherited or overloaded (which is in line with the fact that we can't inherit from a static class, nor can it inherit from any other class than System.Object)
- It cannot be called directly - it is automatically called by the Common Language Runtime (CLR) when the static class is first used

Tip: other interview questions on this topic:

- "**Can a static class have a constructor defined?**"
  Static class cannot have an instance constructor defined, but it can have a static constructor.

- *"***What is a static constructor?***"*
  It's a special method used to initialize static members of a class.

- "**Is it possible to use the "this" keyword within a static method?**"
  No. "This" refers to the current instance, and there is no instance reference in a static method, as a static method belongs to a type as a whole, not to a specific instance.

- "**How to initialize static fields of a class?**"
  By using the static constructor.

- "**Is it possible to have a static constructor in a non-static class?**"
  Yes. A non-static class can have static members, and they can be initialized in the static constructor. In this case, the static constructor will be called before the instance constructor.

17. What is the purpose of the ternary conditional operator?

Brief summary: It's a shorter syntax for the if-else clause. It evaluates a boolean expression and returns the result of one of the two expressions, depending on whether the boolean expression evaluates to true or false.

We often encounter a code like this:

```
string size;
if (dog.Weight > 25)
{
    size = "big";
}
else
{
    size = "small";
}
```

The **ternary conditional operator** allows us to shorten such code significantly:

```
var size = dog.Weight > 25 ? "big" : "small";
```

So the general pattern for this operator is:

var value = boolean expression ? value if true : value if false;

This is why this operator is called a ternary operator - because it needs **three** operands.

You can of course chain this operator:

```
var size = dog.Weight > 25 ? "big" : dog.Weight < 5 ? "small" : "medium";
```

In this case, the underlined expression is itself calculated using the ternary operator, and its' value will be used if the first boolean expression (dog.Weight > 25) is false.

Please note that we can only use the ternary conditional operator with an assignment. For example, this code would not compile:

```
//the below does not compile because there is no assignment here
dog.Weight > 100 ?
    Console.WriteLine("Wow!") :
    Console.WriteLine("It's a normal dog");
```

Tip: other interview questions on this topic:

- "**Can a ternary operator always be translated to an if-else statement?**"
  Yes. The ternary operator is just a shorter syntax for if-else statements.

- "**Can an if-else statement always be translated to a ternary operator?**"
  No. The ternary operator can only be used with an assignment to a variable, so we can't use it, for example, to call Console.WriteLine method with some argument if the condition is true, and with another, if it is false.

18. What is the purpose of the null coalescing and null conditional operators?

Brief summary: The null coalescing and null conditional operators allow us to perform some operations if a value is null, and others if it's not.

Null coalescing operator and null coalescing assignment operator:

We often see the code like this:

```
class Greeter
{
    public static string Greet(string name)
    {
        if (name == null)
        {
            return "Hello, stranger!";
        }
        return $"Hello, {name}!";
    }
}
```

It is a typical example when we want to do something if the value is not null, and if it is null, we want to do something else - possibly use some kind of default.

The **null coalescing operator** allows us to do this using less code:

```
class Greeter
{
    public static string Greet(string name)
    {
        return $"Hello, {name ?? "Stranger"}!";
    }
}
```

The null coalescing operator returns the left-hand operand if its' value is not null, otherwise, it returns the right-hand operand. The general pattern looks like this:

var result = value ?? Some default is "value" is null

There is also another operator - **null coalescing assignment operator**:

```
private static List<int> _numbers;

static void Main(string[] args)
{
    (_numbers ??= new List<int>()).Add(5);
}
```

This operator will assign a value only if the variable is null.

Null conditional operator:

This operator is used when we want to access a member of an object only if this object is not null.

```
public static void ClearIfNotNull(this List<int?> numbers)
{
    numbers?.Clear();
}
```

In this code, the Clear method from the List class will only be called if the numbers parameter is not null.

Let's consider a bit more complex example:

```
static class ListExtensions
{
    public static int GetAtIndex(this List<int?> numbers, int index)
    {
        if(numbers != null)
        {
            if(numbers[index] != null)
            {
                return numbers[index].Value;
            }
        }
        throw new ArgumentException($"Index {index} not found in the list.");
    }
}
```

This code can be refactored using both null conditional operator and null coalescing operator:

```
static class ListExtensions
{
    public static int GetAtIndex(this List<int?> numbers, int index)
    {
        return numbers?[index] ??
            throw new ArgumentException($"Index {index} not found " +
            $"in the list or value is null.");
    }
```

In this example, we used the null conditional operator (underlined in green) and null coalescing operator (underlined in blue). The element under the index index will only be accessed if the numbers list is not null.

Tip: other interview questions on this topic:

- "**Let's say you have an object and you want to call a method on it, but you are not sure whether this object is null or not. How can you conditionally create this object (if it was null) without using the "if" keyword?**"
  We can use the null coalescing assignment operator, like for example "(numbers ??= new List<int>()).Add(5);".

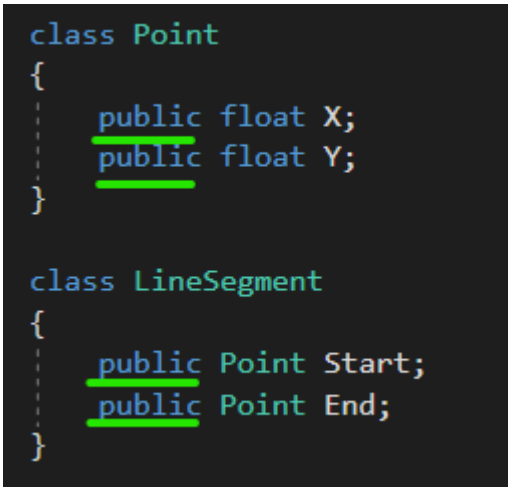
- "**What is the use of the question mark in C#?**"
  It is used in several operators, like ternary operator, null coalescing operator, null coalescing assignment operator, and null conditional operator. Also, we can use it to declare a variable of nullable type.

19. What is encapsulation?

Brief summary: Encapsulation means bundling of data with the methods that operate on that data.

Encapsulation is one of the fundamental concepts of object-oriented programming. **It means bundling of data with the methods that operate on that data.** The term comes from the word "capsule" because we can think of it as the data and the methods are enclosed in a capsule together.

Many people identify encapsulation with data hiding - using private fields instead of public ones. But encapsulation and data hiding are **not** the same things. They work together well, but conceptually they are different. Let's see some code to understand it better:



We defined a line segment, which is a very simple class storing two points. Now, imagine we want to calculate the length of the segment:

```
static void Main(string[] args)
{
    var lineSegment = new LineSegment
    {
        Start = new Point { X = -3, Y = -2 },
        End = new Point { X = 1, Y = 1 },
    };
    Console.WriteLine($"Length of the segment is {Length(lineSegment)}");
    Console.ReadKey();
}

static float Length(LineSegment lineSegment)
{
    var xCoordinatesDifference = lineSegment.End.X - lineSegment.Start.X;
    var yCoordinatesDifference = lineSegment.End.Y - lineSegment.Start.Y;
    return (float)Math.Sqrt(
        (xCoordinatesDifference * xCoordinatesDifference) +
        (yCoordinatesDifference* yCoordinatesDifference));
}
```

As for now, there is no encapsulation here. The data (start and end of the line segment) is separated from the method operating on this data (the Length method). Also, there is no data hiding here, as fields Start, End, X, and Y are public.

Let's introduce encapsulation:

```
class LineSegment
{
    public Point Start;
    public Point End;

    public float Length()
    {
        var xCoordinatesDifference = End.X - Start.X;
        var yCoordinatesDifference = End.Y - Start.Y;
        return (float)Math.Sqrt(
            (xCoordinatesDifference * xCoordinatesDifference) +
            (yCoordinatesDifference * yCoordinatesDifference));
    }
}

class Program
{
    static void Main(string[] args)
    {
        var lineSegment = new LineSegment
        {
            Start = new Point { X = -3, Y = -2 },
            End = new Point { X = 1, Y = 1 },
        };
        Console.WriteLine($"Length of the segment is {lineSegment.Length()}");
        Console.ReadKey();
    }
```

The calculation of the length of the segment has been moved where it belongs - to the LineSegment class.

Please note that now we have encapsulation, but we still don't have data hiding. As we said before, they are not equivalent concepts. We can have encapsulation without data hiding, which is exactly the case we have now. On the other hand, they work well together - if we didn't have encapsulation, we would often be forced to make some fields public so they could be accessed by the methods defined outside their class, the methods that would operate on them.

Let's introduce data hiding now. I will make the Start and End readonly, as well as X and Y in the Point class:

```
class Point
{
    public float X { get; }
    public float Y { get; }

    public Point(float x, float y)
    {
        X = x;
        Y = y;
    }
}

class LineSegment
{
    public Point Start { get; }
    public Point End { get; }

    public LineSegment(Point start, Point end)
    {
        Start = start;
        End = end;
    }
```

We also must adjust the creation of the LineSegment instance:

```
var lineSegment = new LineSegment(new Point(-3, -2), new Point(1, 1));
```

Great. Now we have both encapsulation and data hiding.

To summarize:

- Encapsulation and data hiding are not the same things
- Encapsulation may enable data hiding because if all methods operating on the data would live outside the class that stores the data, this data would have to be made public
- When using encapsulation, it is a good idea to also use data hiding. If the data would be publically accessible, it would be tempting for other programmers to use it outside the class that stores this data. That could lead to code duplication and inconsistency in how the data is operated on - one programmer could define some operation in a different way than another programmer, and then it would be confusing which solution to use. With encapsulation, all users of the class must use the implementation that was defined in that class.

Lastly, let's see **why using encapsulation is a good idea:**

- It makes code more logical, as operations on data are defined in the same place as the data itself
- The operations are defined only once (assuming the data is also hidden)
- The implementation details are hidden from the users - they rely on an interface. That makes the code changes easier because when we need to we can switch the implementation to something else, and it will not affect the users of this class

Tip: other interview questions on this topic:

- "**Is encapsulation the same thing as data hiding?**"
  No. Encapsulation means storing data and methods operating on this data in a single class. Data hiding is about making members of a class non-public.

- "**What is the difference between encapsulation and abstraction?**"
  Abstraction is about generalization - we create abstract types that then can be made more specific in derived types. Encapsulation means storing data and methods operating on this data in one class. We can have a completely non-abstract type that is encapsulated.

20. What is LINQ?

Brief summary: LINQ is a set of technologies that allow simple and efficient querying over different kinds of data.

Data can be stored in various types of containers - C# data structures like Lists or arrays, databases, XML documents, and many more. LINQ allows us to query such data in a uniform way, allowing the programmer to focus on what the program is supposed to do with the data, not on the technical details of accessing different data containers. We can use different LINQ providers, like **LINQ to SQL** that allows querying over SQL databases, or **LINQ to XML** that allows querying over XML documents. Each LINQ provider must implement IQueryProvider and IQueryable interfaces. We can create our own LINQ providers if we need to support querying over a new type of data container.

Let's try to use LINQ in practice. We will work on the following data:

```
var pirates = new List<Person>
{
    new Person("Anne", "Bonny", 1698),
    new Person("Charles", "Vane", 1680),
    new Person("Mary", "Read", 1690),
    new Person("Bartolomew", "Roberts", 1682),
};
```

LINQ allows us to use two alternative ways of querying over data:

- **Query syntax**:

```
var bornAfter1685QuerySyntax = from pirate in pirates
                    where pirate.YearOfBirth > 1685
                    select pirate;
```

- **Method syntax**:

```
var bornAfter1685MethodSyntax = pirates.Where(pirate => pirate.YearOfBirth > 1685);
```

Both expressions do the same thing - they filter out those pirates who were born after 1685. There isn't any distinct advantage of one over the other. Any query syntax can be transformed into method syntax.

From my personal experience, most developers prefer method syntax, as it is just pure C#, over query syntax which is kind of a new language. But it is up to you (or your team) which one you should use.

Let's see some of the most useful methods from the System.LINQ namespace. From now on I'm going to stick to method syntax.

```
IEnumerable<Person> orderedByLastName = pirates.OrderBy(pirate => pirate.LastName);

IEnumerable<int> onlyYearsOfBirth = pirates.Select(pirate => pirate.YearOfBirth);

double averageYearOfBirth = pirates.Average(pirate => pirate.YearOfBirth);

bool isAnyPirateBornBefore1650 = pirates.Any(pirate => pirate.YearOfBirth < 1650);

bool areAllPiratesBornAfter1650 = pirates.All(pirate => pirate.YearOfBirth > 1650);

IEnumerable<Person> piratesWithLastNameStartingWithR = pirates.Where(
    pirate => pirate.LastName.StartsWith("R"));

Person firstPirateByAlphabet = pirates.OrderBy(pirate => pirate.LastName).First();

IEnumerable<Person> piratesFromYoungestToOldest = pirates.OrderBy(
    pirate => pirate.YearOfBirth).Reverse();
```

As you can see, those expressions are really easy to read and understand. LINQ is widely considered an amazing library with intuitive syntax that was well designed by its developers. If you are not familiar with LINQ, I highly recommend you to start learning it, as it may be one of the most powerful tools .NET developers use.

Tip: other interview questions on this topic:

- "**What are the benefits of using LINQ?**"
  It offers a common syntax for querying any type of data source. It provides a simple yet powerful way of manipulating data. LINQ methods are chainable, which means you can have a single expression with multiple LINQ methods. It allows writing cohesive, readable, and flexible code.

- "**What is a LINQ provider?**"
  LINQ provider is any class that implements IQueryProvider and IQueryable interfaces. LINQ providers are used for querying over a particular source of data. Examples of LINQ providers may be LINQ to SQL or LINQ to XML.

21. What are extension methods?

Brief summary: An extension method is a method defined outside a class, that can be called upon this class's objects as a regular member method. Extension methods allow you to add new functionality to a class without modifying it.

Extension methods allow us to add new functionality to a class without actually modifying it, or without adding a new class inheriting from it. We define them in a special static class, and because of that, you don't need to recompile the original class. Even if they are defined in a separate class, you can still call them as regular member methods of the extended class.

Let's see this in practice. Imagine your code operates on long, multiline strings. You need a method that, given a string, counts a number of lines. Unfortunately, there is no such method in the built-in String class. We must create our own method.

```
var multilineString = @"Said the Duck to the Kangaroo,
                        Good gracious! how you hop
                        Over the fields, and the water too,
                        As if you never would stop!
                        My life is a bore in this nasty pond;
                        And I long to go out in the world beyond:
                        I wish I could hop like you,
                        Said the Duck to the Kangaroo.";

var numberOfLines = GetNumberOfLines(multilineString);
```

Something like that. But where to define the new GetNumberOfLines method? Ideally, it would belong to some static class aggregating all string operations. Let's do it:

```
public static class StringOperations
{
    public static int GetNumberOfLines(string multilineString)
    {
        return multilineString.Split("\n").Length;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var multilineString = @"Said the Duck to the Kangaroo,
                                Good gracious! how you hop
                                Over the fields, and the water too,
                                As if you never would stop!
                                My life is a bore in this nasty pond;
                                And I long to go out in the world beyond:
                                I wish I could hop like you,
                                Said the Duck to the Kangaroo.";

        var numberOfLines = StringOperations.GetNumberOfLines(multilineString);
```

All right - this code works, but it seems a bit clumsy. Over time, more and more methods could join the StringOperations class, and each of them would have to be called with StringOperations.MethodName syntax. Wouldn't it be better if we could simply call "multilineString.NumberOfLines()"? Unfortunately, we can't modify the String class. But we can use extension methods, and that's exactly their purpose. Let's do it:

```
public static class StringExtensions
{
    public static int NumberOfLines(this string input)
    {
        return input.Split("\n").Length;
    }
}

class Program
{
    static void Main(string[] args)
    {

        var multilineString = @"Said the Duck to the Kangaroo,
                                Good gracious! how you hop
                                Over the fields, and the water too,
                                As if you never would stop!
                                My life is a bore in this nasty pond;
                                And I long to go out in the world beyond:
                                I wish I could hop like you,
                                Said the Duck to the Kangaroo.";

        var numberOfLines = multilineString.NumberOfLines();
```

Now the NumberOfLines method is an extension method for the String class. Please note "this" before the parameter type. We need to use the "this" keyword in order to create an extension method. It also must be static and must belong to a static, non-generic class. Notice how we call this method now - exactly like it belonged to the String class.

I also renamed StringOperations to StringExtensions as this is the more conventional way of naming the classes containing extension methods.

As you can see, the extension method is an easy way to add functionality to a class without modifying it. It's most commonly used with external classes (not defined in our project, so built-in classes or classes from external libraries) but we can also use it with our own classes:

```
public class Duck
{

}

public static class DuckExtensions
{
    public static string Quack(this Duck duck)
    {
        return "Quack, quack, I'm a duck";
    }
}
```

If the class already contains a method with the same signature as the extension method, the member method will be called and the extension method will be ignored:

```
public class Duck
{
    public string Quack()
    {
        return "(Not an extension method) Quack, quack, I'm a duck";
    }
}

public static class DuckExtensions
{
    public static string Quack(this Duck duck)
    {
        return "(This is an extension method) Quack, quack, I'm a duck";
    }
}
```

Please note that both of those methods would be called upon a duck object with "duck.Quack()". In this case, calling the Quack method on a duck object will result in:

```
(Not an extension method) Quack, quack, I'm a duck
```

As you can see the non-extension method has been called, as it has a priority over extension methods.

To summarize: extension methods allow us to add functionality to a class without modifying it, which is especially useful when we use an external class and we don't have access to its source code. Extension methods are used very often and you might even have used them without knowing it - for example, whenever calling LINQ's methods OrderBy, GroupBy, or Where, you are actually calling extension methods for IEnumerable. Here is a snippet from LINQ source code to prove it:

```
public static partial class Enumerable
    {
        public static IEnumerable<TSource> Where<TSource>(
            this IEnumerable<TSource> source, Func<TSource, bool> predicate)
            {
```

Requirements:

- The class in which the extension methods are defined must be static and non-generic
- The extension method must take the object of the extended class as a first parameter, with "this" modifier proceeding this parameter

Tip: other interview questions on this topic:

- "**How would you add new functionality to an existing class, without modifying this class?**"
  By using extension methods.

- "**What will happen if you call a member method that has the same signature as the existing extension method?**"
  The member method has priority and it will be the one to be called. The extension method will not be called.

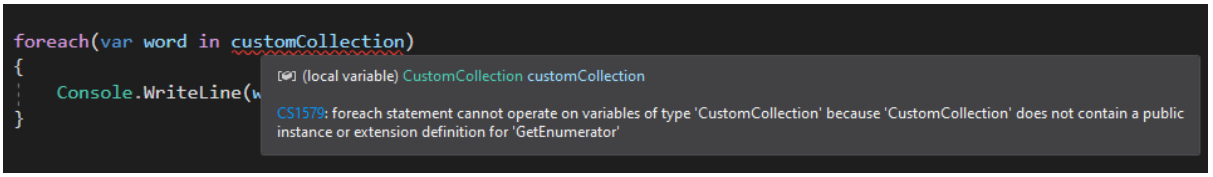
22. What is IEnumerable?

Brief summary: IEnumerable is an interface that enables iterating over a collection with a foreach loop.

IEnumerable is an interface that enables iterating over a collection with a **foreach** loop:

```
var words = new[] { "a", "little", "duck" };

foreach(var word in words)
{
    Console.WriteLine(word);
}
```

The above code works because the array we used implements IEnumerable interface. If it hadn't, the code would not compile:



There is also a generic counterpart of IEnumerable interface - IEnumerable<T> that can hold any type of item. The important thing to note here is that LINQ provides a lot of extension methods for operating on IEnumerable<T> objects. Those methods can be used for ordering, filtering, or aggregating the data, and many more. For simplicity, we will focus on non-generic IEnumerable.

Another important feature of IEnumerable interface is that it provides **read-only** access to a collection - it doesn't expose any methods that allow modification of the collection. When creating a method returning a collection it is recommended to return it as a read-only collection unless we really want to let the user of this method modify the collection.

All right. Let's take a closer look at this interface. It contains a single method:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

As you can see, GetEnumerator simply returns IEnumerator:

```
public interface IEnumerator
{
    bool MoveNext();
    Object Current { get; }
    void Reset();
}
```

An **enumerator** is a mechanism that allows iterating over collection elements. You can imagine it as a kind of a pointer that points to a "current" element in the collection (that element can be accessed with the Current property). To iterate forward, the MoveNext method is executed. It moves the pointer one element up, assuming we are not already at the end of the collection. If we are, MoveNext will return false. There is also the Reset method, which moves the pointer back to the beginning of the collection.

We will implement our own type supporting enumeration in a second, but first, let's see how foreach loop is interpreted by .NET. Let's see the code with a foreach loop again:

```
var words = new[] { "a", "little", "duck" };

foreach (var word in words)
{
    Console.WriteLine(word);
}
```

.NET actually translates this code to something like this:

```
IEnumerator wordsEnumerator = words.GetEnumerator();
string word;
while (wordsEnumerator.MoveNext())
{
    word = (string)wordsEnumerator.Current;
    Console.WriteLine(word);
}
```

Now it is obvious why we need to implement IEnumerable to support usage of foreach loops - if we didn't, the above code would simply not compile.

All right, let's create our own collection that will hold a group of strings, and that will support being iterated over by foreach loop. To do so, we need to create a class that implements IEnumerable interface:

```
class WordsCollection : System.Collections.IEnumerable
{
    private string[] _words;

    public WordsCollection(string[] words)
    {
        _words = words;
    }

    public IEnumerator GetEnumerator()
    {
        throw new NotImplementedException();
    }
}
```

Now, we will have to create the WordsEnumerator class that will implement IEnumerator interface:

```
class WordsCollection : System.Collections.IEnumerable
{
    private string[] _words;

    public WordsCollection(string[] words)
    {
        _words = words;
    }

    public IEnumerator GetEnumerator()
    {
        return new WordsIterator(_words);
    }
}

class WordsIterator : IEnumerator
{
    private string[] _words;

    public WordsIterator(string[] words)
    {
        _words = words;
    }

    public object Current => throw new NotImplementedException();

    public bool MoveNext()
    {
        throw new NotImplementedException();
    }

    public void Reset()
    {
        throw new NotImplementedException();
    }
}
```

As you remember, we said that an enumerator is kind of a pointer to a current element. We must store the element this pointer points to as a private field. We will use an integer named _position that will refer to the index in the _words array:

```
class WordsIterator : IEnumerator
{
    private string[] _words;
    private int _position = -1;
```

You may be surprised why we used -1, not 0. To understand it, let's see again how .NET interprets the foreach loop:

```
IEnumerator wordsEnumerator = words.GetEnumerator();
string word;
while (wordsEnumerator.MoveNext())
{
    word = (string)wordsEnumerator.Current;
    Console.WriteLine(word);
}
```

As you can see, the MoveNext method is called **first**, and **then** we access the Current element. The MoveNext method will be incrementing the value of _position field by one. If we initialized the _position field with 0, it would have a value of 1 after the MoveNext method was called. And because of that, the "first" element returned by the Current property would actually be the **second** in the collection. This is why we set the position to -1, so after the first use of MoveNext it is 0.

All right, now it is pretty obvious what the Reset method will do:

```
public void Reset()
{
    _position = -1;
}
```

MoveNext will simply increment the _position by one. Please note that MoveNext returns a bool. This bool should be true if we successfully advanced to the next element, and false if we passed the end of the collection. Let's implement that:

```
public bool MoveNext()
{
    _position++;
    return _position < _words.Length;
}
```

Finally, let's implement the Current property:

```
public object Current
{
    get
    {
        try
        {
            return _words[_position];
        }
        catch (IndexOutOfRangeException)
        {
            throw new InvalidOperationException("Collection end reached");
        }
    }
}
```

That's it! Now it works correctly with the foreach loop:

```
var wordsCollection = new WordsCollection(words);
foreach (var word in wordsCollection)
{
    Console.WriteLine(word);
}
```

All right, let's summarize what we learned about IEnumerable:

- It allows looping over collections with a foreach loop
- It works with LINQ query expressions
- It allows read-only access to a collection
- To implement it, we must create an enumerator class, that will provide MoveNext and Reset methods, as well as Current property

Tip: other interview questions on this topic:

- "**What is an enumerator?**"
  An enumerator is a mechanism that allows iterating over collection elements. It's a kind of a pointer that points to a "current" element in the collection.

- "**Assuming a method returns a collection of some kind, how to best express your intent if you don't want the user to modify this collection?**"
  By returning it as IEnumerable or another readonly collection type.

23. What is the difference between the equality operator (==) and Equals?

Brief summary: In the most common scenario == compares objects by reference while Equals is overridden to compare them by content. Both can have custom implementation for any type, so this behavior may vary.

Before we explain the difference between the == operator and the Equals method we must understand couple of important things:

- Equals is a method from System.Object class, so every object inherits it
- each class may have the == operator overloaded, as well as the Equals method overridden. That means, we can't define the universal difference between those two methods of checking equality, because for each type it may be different

First, let's talk about the **reference types**.

The general rule is that:

- == operator compares references
- The Equals method should be overridden if we want to define the custom way of comparing objects. Still, if not overridden, it will compare objects by reference, just like the == operator.

The important thing is that the == operator can also be overloaded, so we can't always assume that it will compare objects by reference for any type.

Okay, let's take a look at a simple type that only wraps an integer variable. First, let's see a version that does not override the Equals method:

```
class CustomClass
{
    int _x;
    public CustomClass(int x)
    {
        _x = x;
    }
}
```

Let's create two identical objects of this class and use both == operator and Equals method to compare them:

```
Console.WriteLine("For CustomClass:");
var customClass1 = new CustomClass(1);
var customClass2 = new CustomClass(1);
Console.WriteLine($"customClass1 == customClass2: {customClass1 == customClass2}");
Console.WriteLine($"customClass1.Equals(customClass2): {customClass1.Equals(customClass2)}\n");
```

The result of this comparison will be **false** for **both** cases:

- == operator compares references. We have two separate objects so their references differ
- Equals method is not overridden, so the Equals method from the Object class is called, and it also compares objects by references.

Now, let's have a class that overrides the Equals method:

```
class CustomClassOverridingEquals
{
    int _x;
    public CustomClassOverridingEquals(int x)
    {
        _x = x;
    }

    public override bool Equals(object obj)
    {
        var item = obj as CustomClassOverridingEquals;

        if (item == null)
        {
            return false;
        }

        return _x == item._x;
    }

    public override int GetHashCode()
    {
        return _x.GetHashCode();
    }
}
```

You might have noticed that I have also overridden the GetHashCode method. I don't want to get into details about why I did so, because this is beyond junior level. Let me just state that this is not necessary for the Equals method to work, but it is a recommended practice to override the GetHashCode method when we override the Equals method.

Let's compare those objects:

```
Console.WriteLine("For CustomClassOverridingEquals:");
var customClassOverridingEquals1 = new CustomClassOverridingEquals(1);
var customClassOverridingEquals2 = new CustomClassOverridingEquals(1);
Console.WriteLine($"customClassOverridingEquals1 == customClassOverridingEquals2: " +
    $"{customClassOverridingEquals1 == customClassOverridingEquals2}");
Console.WriteLine($"customClassOverridingEquals1.Equals(customClassOverridingEquals2): " +
    $"{customClassOverridingEquals1.Equals(customClassOverridingEquals2)}\n");
```

Now, the == operator returned false, but overridden Equals returned true. This is the most typical implementation of the equality comparison for objects: == comparing references and Equals comparing values.

We could of course overload the == operator to also compare values. This is how it is done for the string class. That's why when comparing strings with both == operator and Equals method, the values will be compared:

```
Console.WriteLine("For strings:");
string string1 = "abc";
string string2 = "abc";
Console.WriteLine($"string1 == string2: {string1 == string2}");
Console.WriteLine($"string1.Equals(string2): {string1.Equals(string2)}\n");
```

In this case, **both** operations will return **true**.

Let's talk about **value types** now. Value types do not have references, so what happens when we compare them? Let's create a struct (remember, the struct is a value type) similar to the classes defined above:

```
struct CustomStruct
{
    int _x;
    public CustomStruct(int x)
    {
        _x = x;
    }
}
```

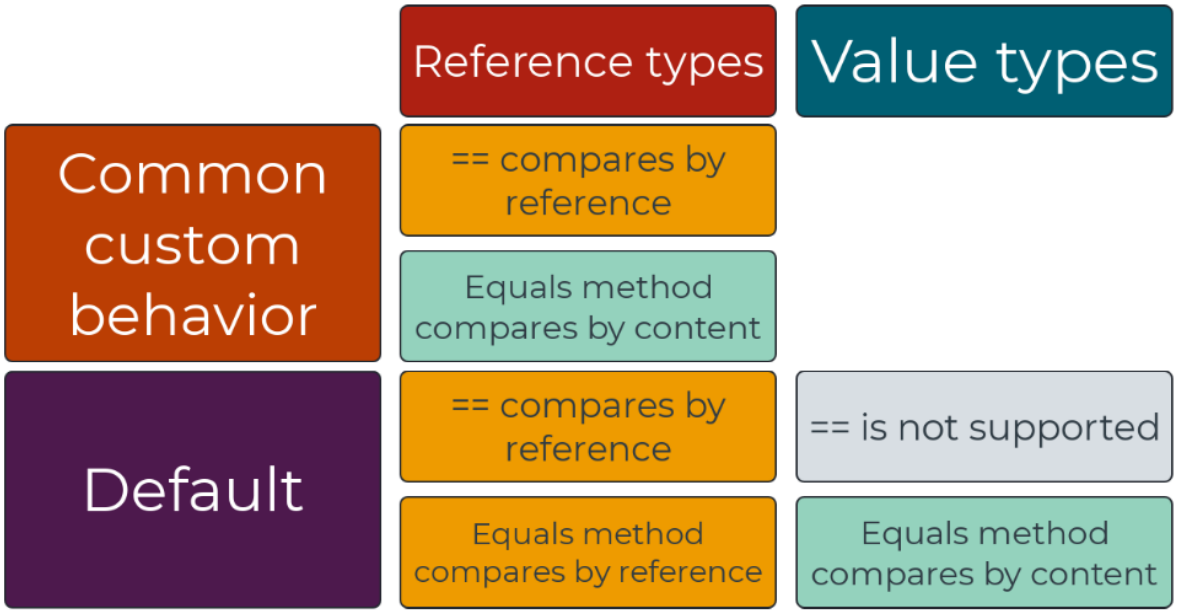
And now let's try to compare two structs identical by value:

```
Console.WriteLine("For CustomStruct:");
var customStruct1 = new CustomStruct(1);
var customStruct2 = new CustomStruct(1);
Console.WriteLine($"CustomStruct1 == CustomStruct2: {customStruct1 == customStruct2}");
Console.WriteLine($"CustomStruct1.Equals(CustomStruct2): {customStruct1.Equals(customStruct2)}\n");
```

First of all, the usage of the == operator leads to a compilation error. By default, structs do not support this operator. If we want to use it, we must overload it.
The comparison with the Equals method returns true - this is because the default implementation of this method for value types reads all fields belonging to the type and compares them by value one by one. In this case, we have one field of type int, and it was of value 1 for both customStruct variables.

That makes structs pretty handy when we want to support comparison by value without overriding the Equals method. On the other hand, the default implementation uses reflection, which is not very good from the performance point of view - that's why it is often recommended to override Equals method for structs by hand (Reflection is a topic beyond junior level, so I will skip the detailed explanation of what it is).

And here is a simple summary:



Tip: other interview questions on this topic:

- "**What's the difference between comparing equality by reference and by value?**"
  Comparing by reference simply checks if two variables point to the same object in memory. Comparing by value checks the internal value or values of an object. For example, we can have two objects of type Point, both having X = 10 and Y = 20. Even if they are two separate objects and they live in different places in the computer's memory, the comparison by value will consider them equal, while the comparison by reference will not.

- "**Is it possible to use the == operator on structs?**"
  It is by default not supported, but we can use it if we overload it.

24. What is the difference between deep copy and shallow copy?

Brief summary: When creating a shallow copy, value type members will be copied, but only a reference will be copied for reference types. For deep copying also the reference types will be copied into new objects.

Before we understand the difference between deep and shallow copy, **make sure you understand the difference between value types and reference types**. See the "What is the difference between value types and reference types?" lecture for more information.

The need for copying an existing object is very common in the software development process. There are two strategies for copying an object.

- Creating a **shallow copy**. That means, value type members will indeed be copied, but for reference types only a reference will be copied - and it will still point to the same object in memory.
- Creating a **deep copy**. In this case, not only value types will be copied, but also for reference types brand-new objects will be created. The references will not be copied, but a new reference to the new object will be created.

To understand it better let's see two simple classes:

```
class Person
{
    public string Name;
    public int Height;
    public Pet Pet;

    public Person(string name, int height, Pet pet)
    {
        Name = name;
        Height = height;
        Pet = pet;
    }
}
```

```
class Pet
{
    public string Name;
    public int Age;

    public Pet(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

The Person class contains a member of type Pet. Pet is a class, so a reference type. This means, in the person object a **reference** to a pet object will exist. Let's add a ShallowCopy method to the Person class:

```
class Person
{
    public string Name;
    public int Height;
    public Pet Pet;

    public Person(string name, int height, Pet pet)
    {
        Name = name;
        Height = height;
        Pet = pet;
    }

    public Person ShallowCopy()
    {
        return (Person)MemberwiseClone();
    }
}
```

This method uses the MemberwiseClone built-in method that belongs to the Object class. Remember, in C# each class derives from the basic Object type, so it inherits all its methods including MemberwiseClone. MemberwiseClone does exactly what we need - it creates a shallow copy. I don't want to go into details of how this method works under the hood, but you can imagine it like this:

```
public object MemberwiseClone()
{
    return new Person
    {
        Name = this.Name,
        Height = this.Height,
        Pet = this.Pet,
    };
}
```

From the lecture on value types and reference types we know that when we assign one value type variable to another (like, in this example, Height), the value is copied. In the case of reference types (in this case - Pet) only a reference is copied. That means, both the original Person object and its' copy will point to the same Pet object. If the Pet belonging to the copy of a Person object will be modified, it will also affect the original Person's Pet. Let's see this in action:

```
var john = new Person("John", 175, new Pet("Lucky", 5));
var johnShallowCopy = john.ShallowCopy();
johnShallowCopy.Pet.Age = 10;
Console.WriteLine($"John's pet age: {john.Pet.Age}");
Console.WriteLine($"John's shallow copy's pet age: {johnShallowCopy.Pet.Age}\n");
```

What do you think will be printed to the console? Remember, both john and johnShallowCopy have the reference to the same Pet object. When pet's age is modified for johnShallowCopy, it will also affect the pet belonging to john:

```
John's pet age: 10
John's shallow copy's pet age: 10
```

Would the same thing happen if we modified the Height property on johnShallowCopy object?

```
johnShallowCopy.Height = 150;
Console.WriteLine($"John's height: {john.Height}");
Console.WriteLine($"John's shallow copy's height: {johnShallowCopy.Height}\n");
```

Let's see the result:

```
John's height: 175
John's shallow copy's height: 150
```

As you can see, the change in johnShallowCopy did not affect the john object. This is because Height is of type int, and int is a value type. It has been copied when the shallow copy was created.

Now, let's add a DeepCopy method to the Person class:

```
public Person DeepCopy()
{
    return new Person(Name, Height, new Pet(Pet.Name, Pet.Age));
}
```

This time we explicitly create a brand-new Pet object, using the original's Pet data. With a new object comes a new reference. This means the new Person object will have its own reference pointing to a different Pet object in memory. Any changes to the copied object will not be reflected in the original object. Let's see this in code:

```
var mary = new Person("Mary", 165, new Pet("Tiger", 7));
var maryDeepCopy = mary.DeepCopy();
maryDeepCopy.Pet.Age = 11;
Console.WriteLine($"Mary's pet age: {mary.Pet.Age}");
Console.WriteLine($"Mary's shallow copy's pet age: {maryDeepCopy.Pet.Age}");
```

What will be printed to the console this time?

```
Mary's pet age: 7
Mary's shallow copy's pet age: 11
```

Unlike before, the original object's Pet age is not modified. The change in the value type member, for example Height, will give exactly the same result as it did for the shallow copy scenario.

There is one more thing that causes a lot of confusion. Let's consider the following code:

```
var person = new Person("Alex", 183, null);
var person2 = person;
```

People (including interviewers!) often ask "Is this a deep copy or a shallow copy?". The answer is: it is neither. This is simply an assignment. The person2 variable will

simply point to the same object in memory as the person variable does. No members are copied, not even those of value types. That means, if the Height was changed for the person2, it will also be changed for the person:

```
var person = new Person("Alex", 183, null);
var person2 = person;
person2.Height = 160;
Console.WriteLine($"person's height: {person.Height}");
Console.WriteLine($"person2's height: {person2.Height}\n");
```

The result of this code is:

```
person's height: 160
person2's height: 160
```

Remember - both shallow copy and deep copy create a new Person object. In the case of a shallow copy, the reference type members for the original and the copied objects will point to the same objects in memory, whereas in the case of deep copy new references will point to new objects. For the assignment, no copy of the Person object will be created. There still will be only one Person object stored in memory, and we will just have two variables holding references to this object.

One last thing to consider is **when to use shallow and deep copies**. In general - if you need to create a truly independent copy of an object, that you will be able to modify to your liking without affecting the original object, you should create a deep copy. Otherwise, you may consider using a shallow copy.

Tip: other interview questions on this topic:

- "**What does the MemberwiseClone method do?**"
  It creates a shallow copy of an object.

- *"***What's the risk of using shallow copies?***"*
  The risk is that both the original object and the copy can hold references to the same objects. That means, when modifying the object referenced by the original, also the object referenced by the copy will be affected.

25. What is the Garbage Collector?

Brief summary: The Garbage Collector is a mechanism that manages the memory used by the application. If an object is no longer used, the Garbage Collector will free the memory it occupies. The Garbage Collector is also responsible for defragmenting the application's memory.

The **Garbage Collector** is the CLR's (Common Language Runtime) mechanism that manages the memory used by the application.

In some languages (like, for example, C) it was a developer's responsibility to allocate and deallocate the memory needed by objects. It created a lot of noise in the code and was a common source of errors.

In many modern languages, including C#, memory management is built-in and we as developers do not need to worry about it (most of the time).

As soon as the application starts it begins to create new objects, and those objects need to be stored in memory. **C#'s objects live in two areas of memory: the stack and the heap.**

The stack holds value types, so types like ints, bools, floats, and structs. Memory allocated for those objects is automatically freed when the control reaches the end of the scope those objects live in:

```
if(args != null)
{
    int a = 5;

    //end of the scope for the variable "a" - it will be cleaned up right away
}
```

When variable a was created, it was put on the stack. When the control reaches the end of the scope this variable lives in - so the "if" clause - this variable is removed from the stack and its memory is freed. This is NOT done by the Garbage Collector, but by the CLR itself.

Garbage Collector manages objects that live on **the heap**, so the reference types (strings, Lists, arrays, and all other objects that are defined as classes). Garbage

Collector determines if there are any existing references to the object - if not, it decides this object is no longer used and frees the memory used by this object.

```
if(args != null)
{
    string text = " abc";

    //end of the scope for the variable "text" - Garbage Collector will clean it up, but not immediately
}
```

The important thing here is that Garbage Collector **will not clean the memory immediately** - it is important to know that we can't deterministically say when exactly will that happen (so if you **must** clean up some resources at once, don't leave that to Garbage Collector - implement IDisposable interface and use the Dispose method). There is a way to force collection of the memory by the Garbage Collector by using **GC.Collect** method, but it shouldn't be the default solution if you want to have the memory immediately freed.

So when does the Garbage Collector decide to clean up memory?

- When the system has low physical memory (the operating system notifies about that)
- When the memory that's used by allocated objects on the heap surpasses a given threshold. This threshold is continuously adjusted as the process runs
- When the GC.Collect method is called

So basically, the Garbage Collector does not start to work unless there is a need for it.

The important thing to understand is that **Garbage Collector runs on its own, separate thread**, and as this happens all other threads are being **stopped** until Garbage Collector finishes its work. This might obviously cause performance issues. For example, consider a video game created in C# (in case you don't know, one of the most popular video games development environments is Unity, where you can create games in C#). If plenty of short-lived objects would be created every second, Garbage Collector would have a lot of work to do, and it would often "freeze" the game for a fraction of a second to do its work. The experience for the player would not be perfect. In such cases, it is recommended to avoid frequent triggering of the Garbage Collector work (there are some techniques to do it, for example by using a pool of objects that are being reused, instead of frequent creation and destruction of short-lived objects). The important thing to remember is that Garbage Collector freezes all other threads for the time of the collection.

After Garbage Collector finishes freeing the memory, it also executes **memory defragmentation**.

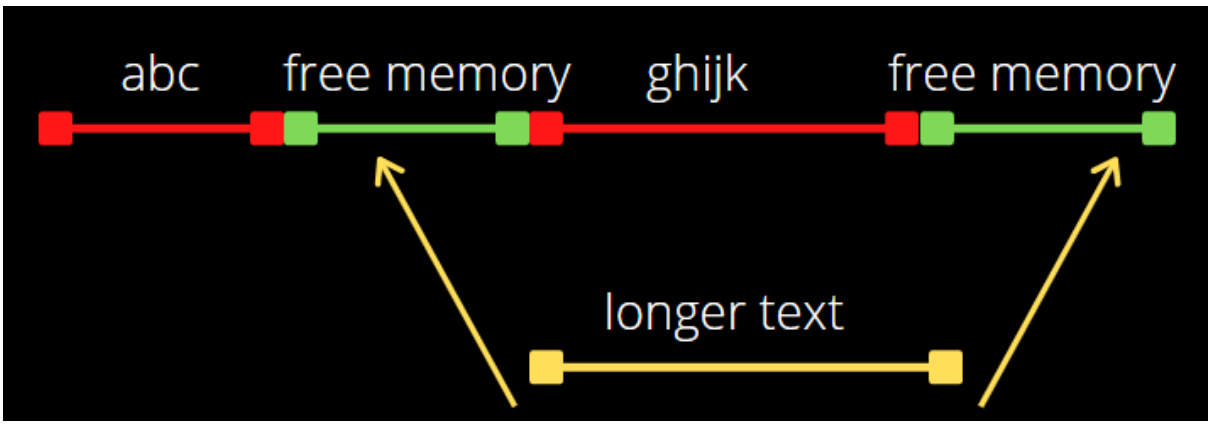
To understand what it is, imagine a computer's memory as a long array of bits (that's actually pretty close to the real thing). Let's say we create a variable of type string, and value of "abc". It consists of 3 characters, and each char in C# is 2 bytes. One byte is 8 bits. So we need 3 * 2 * 8 bits of memory, so 48 bits.

The Garbage Collector finds a 48-bits long part of empty memory in this long array of memory assigned to the process in which our application runs, and reserves it for the "abc" string. Now the memory looks like this:

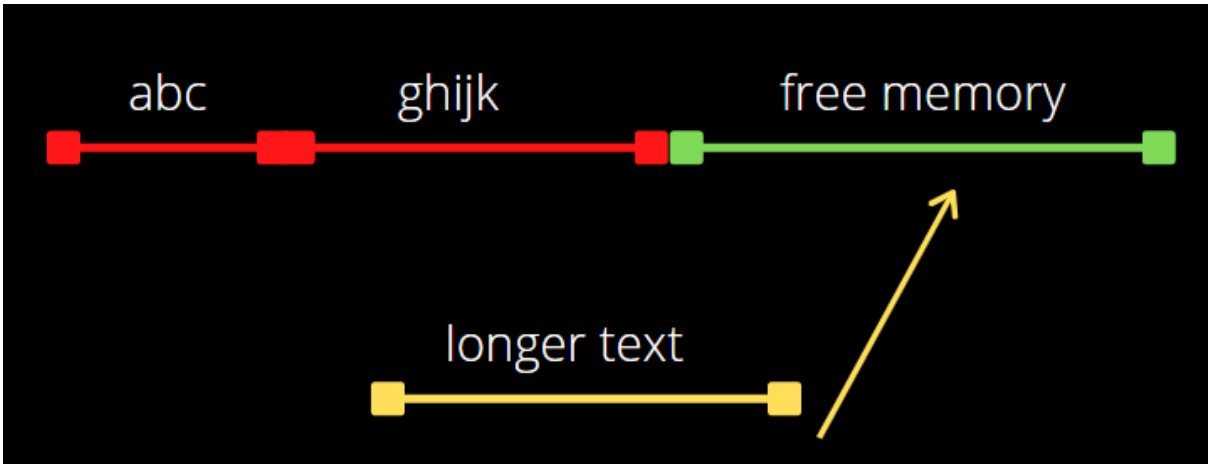
Let's say some other strings had been created:

Now, let's assume the "def" string is no longer used and its memory is freed by the Garbage Collector.

Now, if we want to add some new, long string to the memory, we might actually not find a place for it to fit. We do have two blocks of free memory, but none of them is long enough.



What the Garbage Collector needs to do, is to move some pieces of memory and make them contiguous, thus creating a bigger free block of memory:



Now the new, long string can easily fit in:

This process is called **memory defragmentation**.

So as we can see, the Garbage Collector is a pretty smart tool. It makes our work much easier, but it doesn't mean we can forget completely about memory issues. Let's consider **memory leaks**. A memory leak is a situation when some piece of memory is not being cleaned up, even if the object using it is no longer in use. It is important to understand that **Garbage Collector does not give us 100% protection from memory leaks**.

One of the most common sources of memory leaks in .NET is related to event handlers. Events are topics beyond Junior level, but let's for now just assume they are used to handle some specific situations (like clicking on a button) by "attaching" a method that will be called to the event "handler".

Imagine you have a window-based application. There is the main window, and when a button in this window is clicked, a child window opens. Here is a very simplified code for that:

```
public class MainWindow
{
    public event EventHandler<EventArgs> mainWindowEventHandler;

    private void buttonClick_OpenChildWindow(object sender, EventArgs e)
    {
        var childWindow = new ChildWindow(this);
        childWindow.Show();
    }
}
```

```
public class ChildWindow
{
    private MainWindow _mainApplicationWindow;

    public ChildWindow(MainWindow mainApplicationWindow)
    {
        _mainApplicationWindow = mainApplicationWindow;
        _mainApplicationWindow.mainWindowEventHandler += HandleEventFromMainWindowInChildWindow;
    }

    private void HandleEventFromMainWindowInChildWindow(object sender, EventArgs e)
    {
    }

    internal void Show()
    {
        //opening a window
    }
}
```

Note what happens: when MainWindow's button is clicked, a ChildWindow is being created. We want something to happen in the ChildWindow when an event happens in the MainWindow. In its constructor, the ChildWindow subscribes to the MainWindow's event handler. This is a very important thing: by subscribing to the event handler, a reference from MainWindow to ChildWindow is created. It is necessary because if the event will be triggered in the MainWindow, the ChildWindow must be notified about that.
Now, let's say the user closes the ChildWindow. It would seem like the object of ChildWindow would be cleaned up by the Garbage Collector. But it will not be!

Because there is still a "hidden" reference from the MainWindow to the ChildWindow. Garbage Collector sees this reference and decides that the ChildWindow object is still in use. Now, imagine what will happen if the user keeps opening and closing the ChildWindow. New objects will be created, but they will never be freed. Finally, we will run out of memory.

In this case, the solution is to unsubscribe from the event handler - for example on form closing:

```
void OnWindowClosing()
{
    _mainApplicationWindow.mainWindowEventHandler -= HandleEventFromMainWindowInChildWindow;
}
```

This was just an example of how we can encounter memory leaks that GarbageCollector is unable to protect us against. You should always be considerate of the memory and remember that Garbage Collector is just a tool, even if pretty clever.

There are many low-level details on how exactly Garbage Collector works, but they are beyond Junior level. The most important things you need to remember about Garbage Collector are:

- It manages the memory of the application
- It frees up memory allocated for objects that are no longer referenced
- It's hard to say when exactly it will collect the memory - it has its' own mechanism that depends on how much memory is being used
- It defragments the memory
- It does not guarantee protection from memory leaks

Tip: other interview questions on this topic:

- "**Would the Garbage Collector free up the memory occupied by an integer?**"
  No, since an integer is a value type and Garbage Collector only handles memory occupied by reference types. Value types are stored on the stack which has its own mechanism for freeing the memory.

- "**How to manually trigger the Garbage Collector memory collection?**"
  By calling GC.Collect method from System namespace.

- "**What is memory fragmentation and defragmentation?**"
  When pieces of memory are allocated and freed, the memory becomes defragmented, which means free memory is in small pieces rather than in one long piece. This is called memory fragmentation. If there is a need to

put a big object into memory, it might be impossible to find a block of free memory that is long enough. The process of moving the pieces of allocated memory so they stick together, to create a big chunk of free memory is called defragmentation.

- "**What are memory leaks? Does the Garbage Collector guarantee protection from them?**"
  Memory leaks happen when memory is not freed even if an object is no longer used. No, GC does not guarantee protection from them.

26. What are nullable types?

Brief summary: Nullable type is any type that can be assigned a value of null. Nullable<T> struct is a wrapper for a value type allowing assigning null to the variable of this type. For example, we can't assign null to an integer, but we can to a variable of type Nullable<int>.

Nullable type is any type that can be assigned a value of **null**. In C#, all reference types are nullable by default, and value types are non-nullable by default. That means, we can't assign null to an integer variable, but it is fine to assign null to, for example, a List variable, since List is a reference type and int is a value type.

Null represents "lack of value", or "missing value". There are many business cases when such special value is needed, even for value types. For example, imagine you have a collection of people, and each Person has a Height property. If we decide that Height is an integer (so a non-nullable value type) we might encounter a problem - what if we don't know some person's height? We could make a risky assumption that in this case, we would use some special value, like 0 or -1. But then, what if we want to calculate the average height of all people in this collection? Having people of height 160, -1, 185, -1, 170, we would end up with an average value of 102,6, which obviously doesn't make much sense. It's better to represent height as a nullable integer type, and only include non-null values in the average height calculation.

In C#, we can declare value type as nullable with the "?" operator:

```
public int? Height;
```

Using "?" operator is just a short syntax for:

```
public Nullable<int> Height;
```

We can assign null to a nullable:

```
int? nullableNumber = null;
```

Please note that Nullable<T> is a **struct**, which means Nullable is a value type (as all structs are). As we said, we can't assign null to a value type - so how is it possible that we assign a null to a Nullable<T>? Actually, it's a trick of C#'s compiler. Behind the scenes, it interprets the above code as:

```
Nullable<int> nullableNumber = new Nullable<int>();
```

So as you can see, it's not *really* assigning a null.

Nullable exposes useful properties like HasValue, which indicates whether the value is null or not, or Value, which unpacks the internal value (so for the nullable type int? Value property will return an int).

Note: there is also a concept of nullable and non-nullable reference types, that was introduced in C# 8.0. It is not closely related to this lecture, so we will skip the details. In short, it allows a programmer to receive a compiler warning if a reference type has a null value.

Tip: other interview questions on this topic:

- "**Can a value type be assigned null?**" *No, it can't. It must be wrapped in the Nullable<T> struct first if we want to make it nullable.*

- "**Is it possible to have a variable of type Nullable<T> where T is a reference type? For example, Nullable<string>?**"
  No, the Nullable struct has a type constraint on T that requires the T to be a value type. Providing a reference type as T will result in a compilation error.

27. What is a property?

Brief summary: A property is a member that provides a mechanism of reading, writing, or computing a value of a private field.

A **property** is a member that provides a mechanism of reading or writing a value of a private field. To understand properties we must first understand fields. Let's see some simple code:

```
class Person
{
    public int yearOfBirth;

    public Person(int yearOfBirth)
    {
        this.yearOfBirth = yearOfBirth;
    }
}
```

In the above code, the yearOfBirth is a **field**. It simply holds the value that was provided with the constructor call. When using an object of the Person class, you can get and set the value of this field as you like:

```
var person = new Person(1950);
person.yearOfBirth = 2100;
```

Now, let's change this field to a property:

```
class Person
{
    public int YearOfBirth { get; set; }

    public Person(int yearOfBirth)
    {
        YearOfBirth = yearOfBirth;
    }
}
```

This is a subtle yet important change. The YearOfBirth is not a field anymore - it is a **property**, which can be used as if it was a public data member, but actually, it is a special member that exposes two accessors - **get** and **set**. **Accessors** are special methods used to read (get) and write (set) the value of a property. By now, we can still use a property in the same way as we used a field:

```
var person = new Person(1950);
person.YearOfBirth = 2100;
```

To understand better what a property is let's see how this code looks like in older versions of C#. Please be aware that this code is logically equivalent to the one above, this is just an older syntax:

```
private int _yearOfBirth;
public int YearOfBirth
{
    get
    {
        return _yearOfBirth;
    }
    set
    {
        _yearOfBirth = value;
    }
}
```

Now we can see that a property is a kind of wrapper over a field, that exposes methods for getting and setting the value of this field.

By now you might be asking "Okay, but what's the point of using the property if I can achieve exactly the same thing by just using a public field?".

The answer is **encapsulation and data hiding**. When using a field, we are exposing an inner implementation detail of the Person class. To understand why this is a problem, imagine that at some point you (the developer of the application) were told that from now on we need to keep not only the year of birth value, but the full date of birth, including day and month. Let's see the code using a field again:

```
class Person
{
    public int yearOfBirth;

    public Person(int yearOfBirth)
    {
        this.yearOfBirth = yearOfBirth;
    }
}
```

We need to store the date of birth, so we decide to use a DateTime type to store it:

```
class Person
{
    public int yearOfBirth;
    public DateTime dateOfBirth;
```

The problem is that now we have duplicate information about the year of birth. It is stored in the yearOfBirth field as well as in the dateOfBirth. But we can't simply remove the yearOfBirth field - before we exposed it as a public field, and now there can be thousands of places in our application that may be using it. Removing it will generate an immense number of compilation errors. Fixing them will not only be laborious, but it will also put us at risk of causing multiple bugs. So maybe… let's keep both fields? But then, if a user of the Person class wants to know Person's year of birth, she will be confused about which field should be used. If one value is changed, the other should also be - so we should provide some kind of synchronization mechanism. That doesn't sound good. We are in trouble. **By exposing the inner implementation detail of the class we made any changes to this class very risky and hard.**

What would happen if we used a property?

We would start with a class like this:

```
class Person
{
    private int _yearOfBirth;
    public int YearOfBirth
    {
        get
        {
            return _yearOfBirth;
        }
        set
        {
            _yearOfBirth = value;
        }
    }

    public Person(int yearOfBirth)
    {
        YearOfBirth = yearOfBirth;
    }
}
```

Now, we would add a property DateOfBirth of DateTime type:

```
class Person
{
    private int _yearOfBirth;
    public int YearOfBirth
    {
        get
        {
            return _yearOfBirth;
        }
        set
        {
            _yearOfBirth = value;
        }
    }

    private DateTime _dateOfBirth;
    public DateTime DateOfBirth
    {
        get
        {
            return _dateOfBirth;
        }
        set
        {
            _dateOfBirth = value;
        }
    }

    public Person(int yearOfBirth)
    {
        YearOfBirth = yearOfBirth;
    }
}
```

But now, unlike in example with using fields, we can actually safely remove the duplicated field:

```
class Person
{
    public int YearOfBirth
    {
        get
        {
            return _dateOfBirth.Year;
        }
        set
        {
            _dateOfBirth = new DateTime(value, _dateOfBirth.Month, _dateOfBirth.Day);
        }
    }

    private DateTime _dateOfBirth;
    public DateTime DateOfBirth
    {
        get
        {
            return _dateOfBirth;
        }
        set
        {
            _dateOfBirth = value;
        }
    }

    public Person(int yearOfBirth)
    {
        YearOfBirth = yearOfBirth;
    }
}
```

The program will still compile - we kept the YearOfBirth **property**, but we changed its implementation (remember, a property is like a special kind of method - and unlike a field, we can provide an implementation that suits us). Now, getting the value of the YearOfBirth property will actually return the year stored in the DateOfBirth property. Setting the YearOfBirth property will change the year in the DateOfBirth property (leaving the month and the day unchanged).

This is a huge difference. We did not break the application, we don't need to find and fix possibly thousands of places where this property was used. And imagine how much worse it would be if this was a public library, used by people all around the world - if they downloaded a new version of our code, their projects would not compile anymore. That would be a disaster.

So as you can see, using a property instead of a field allowed us to provide a custom implementation of the get and set accessors. It hid the implementation details of the class from the users, allowing us to change it freely and without causing any problems. But this is not the end of the power of properties. Actually, a property does not need to wrap a particular field. It can simply calculate some value:

```
public DateTime? DateOfDeath { get; set; }

public int? LengthOfLife
{
    get
    {
        return DateOfDeath == null ?
            null :
            (int)((DateOfDeath.Value - DateOfBirth).TotalDays / 365);
    }
}
```

In this code, the LengthOfLife property is simply calculated using the DateOfBirth and the DateOfDeath properties.

Another advantage of using properties is that we can define different access right for reading and writing:

```
public string LastName { get; private set; }

private void SetLastName(string lastName)
{
    //it's ok - we can change this private property within the Person class
    LastName = lastName;
}
```

In this case, we can get the value of LastName property anywhere, but we can only set it in the Person class:

```
class Program
{
    static void Main(string[] args)
    {
        var person = new Person(1950, "Smith");
        person.LastName = "Swanson";

    }
}
```

string Person.LastName { get; private set; }

CS0272: The property or indexer 'Person.LastName' cannot be used in this context because the set accessor is inaccessible

Also, we can define any logic on property get or set methods - for example validation:

```
private DateTime _dateOfBirth;
public DateTime DateOfBirth
{
    get
    {
        return _dateOfBirth;
    }
    set
    {
        if (value < DateTime.Now) //we don't allow dates from the future
        {
            _dateOfBirth = value;
        }
    }
}
```

Let's summarize the advantages of using properties over fields:

- Fields expose the inner implementation of a class, properties hide it
- Fields cannot have any custom behavior on reading and writing - properties can
- Fields have the same access modifier for reading and writing - properties can have it different
- Properties can produce results calculated based on other fields or properties
- Properties can be overridden

For all those reasons, it is usually wise to use properties rather than fields. Throughout this course, you've probably seen me using fields more often, but I do it only for brevity and simplicity of the examples. It does not mean I recommend using fields in everyday programming.

Tip: other interview questions on this topic:

- "**What is the difference between a property and a field?**"
  Properties provide a level of abstraction. They work as special methods used to access class-specific data. They do not need to expose fields - they may expose constant literals or some calculated value. Also, properties can be overridden.

- "**How can you encapsulate a field of a class?**"
  By hiding it behind a property.

- "**What are the benefits of using properties?**"
  Encapsulation, custom behavior on reading and writing, different access modifiers for reading and writing. Also, properties can be calculated basing on other data.

28. What are generics?

Brief summary: Generic classes or methods are parametrized by type - like, for example, a List<T> that can store any type of elements.

Generics allow us to create classes or methods that are **parametrized by type**.

Let's consider a simple example of a List. Without generics, we would need to have a separate class for all data types we want to store in a list:

```
public class IntegersList
{
    public void Add(int item)
    {
        //...
    }
}

public class StringsList
{
    public void Add(string item)
    {
        //...
    }
}

public class DoublesList
{
    public void Add(double item)
    {
        //...
    }
}
```

That would be absolutely awful. Not only would we need to copy this code each time we want a new type to be stored in a list, but also if we decided a change must be made in the List classes (for example, if we found a bug) we would need to modify each and every one of them.

This is where generics come in handy - we can create a single class, that will be parameterized by a type:

```
public class List<T>
{
    public void Add(T item)
    {
        //...
    }
}
```

Now, we can store **any** type in the List.

Of course, C# already provides a very good implementation of a List<T> - it can be found in System.Collections.Generic namespace.

We can parametrize a class or a method with more than one type parameter. An example of such a class is a Dictionary. Dictionary is a data structure that holds pairs of keys and values. We can use any types as keys and values:

```
var currencies = new Dictionary<string, string>
{
    ["USA"] = "USD",
    ["Great Britain"] = "GBP"
};

var yearsOfBirth = new Dictionary<string, int>
{
    ["John Smith"] = 1980,
    ["Monica Smith"] = 1983
};
```

In such a case, when defining a generic type, we simply provide two type parameters. Let's see how the C#'s dictionary is defined:

```
public class Dictionary<TKey,TValue>: IDictionary<TKey,TValue>,
```

We sometimes need to put some kind of constraint on a generic type. For example, the Nullable<T> allows only value types to be used as T parameter because non-value types are nullable by definition. To put a constraint on a type we use a "**where"** keyword. Let's see some of the basic type constraints in C#:

Class - the type must be a reference type:

```
public class OnlyReferenceTypes<T> where T : class
{

}
```

Struct - the type must be a value type:

```
public class OnlyValueTypes<T> where T : struct
{

}
```

New() - the type must provide a public parameterless constructor:

```
public class OnlyTypesWithParameterlessConstructor<T> where T : new()
{

}
```

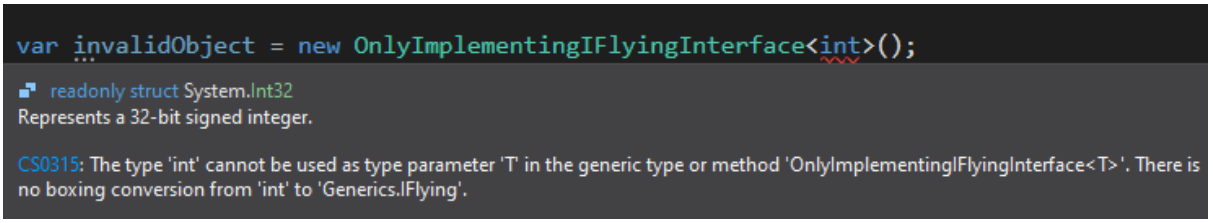
The type must be derived from a base class:

```
public class BaseClass
{

}

public class OnlyDerivedFromBaseClass<T> where T : BaseClass
{

}
```

The type must implement an interface:

```
public interface IFlying
{

}

public class OnlyImplementingIFlyingInterface<T> where T : IFlying
{

}
```

If the type does not meet the criteria defined in the constraint, a compilation error appears:



Tip: other interview questions on this topic:

- "**What are type constraints?**" Type constraints allow limiting the usage of a generic type only to the types that meet specific criteria. For example, we may require the type to be a value type, or require that this type provides a public parameterless constructor.

- "**What is the "where" keyword used for?**" It's used to define type constraints. Also, it is used for filtering when using LINQ.

- "**What are the benefits of using generics?**" They allow us to reduce code duplication by creating a single class that can work with any type. Reducing code duplication makes the code easier to maintain and less error-prone.

29. What is the difference between the "const" and the "readonly" modifiers?

Brief summary:

1. Const fields are assigned at compile time. Readonly fields can be assigned at runtime, in the constructor.
2. Consts can only be numbers, booleans, strings, or a null reference, readonly values can be anything.
3. Consts can't be declared as static, because they are implicitly static. Readonly values can be static.

First, let's take a look at the **const** keyword. It is used to declare a variable whose value is not supposed to change. The exact value must be known at compile time. Const is perfect for declaring variables like, for example:

```
const float PI = 3.14f;
const int DaysInWeek = 7;
const int MaxSizeOfAnArray = 20; //assuming this is the designed limitation
const int BitsInByte = 8;
```

If we don't assign const at declaration, we will get a compiler error:

```
private const int ConstNumber; //must be assigned at declaration
```

Another limitation is that consts can only be numbers, booleans, strings, or a null reference. You can't assign an object that is constructed at runtime to a const - for example, you can't have const Person created like "const Person person = new Person("John", "Smith", 1980);" because the construction of this object happens at runtime, and the value of a const must be determined at compile time.

```
//must be compile-time constant
private const Person ConstPerson = new Person("John", "Smith", 1980);
```

Lastly, consts can't be declared as static, because they are implicitly static. Think about it like that: "static" means that some piece of data belongs to the class as a

whole, not to some particular instance. Since the value of const is always the same, there is no point in having a copy of this data for every instance of the class. It's ok for it to be static, and the only existing copy in the memory can be shared between all instances of the class.

Let's move on to the **readonly** keyword. Readonly means the value will not be reassigned after it has been assigned for the first time. That may seem similar to the const, but there is a difference - the readonly values do not need to be known at the compile time, they can be a result of some calculation or user input. We use readonlys anywhere where we want the value to never change after it was first set.

The readonly member doesn't have to be assigned at declaration - unlike const, it can be assigned via the constructor:

```
private readonly int ReadonlyNumber;

//it is fine to assign readonly number at declaration
private readonly int OtherReadonlyNumber = 4;

public Program() //constructor
{
    //this will not compile because we can only assign consts at declaration
    //ConstNumber = 5;

    //it is fine to assign readonly value in constructor
    ReadonlyNumber = 10;

    //we can also assign a value even if we already did it at declaration
    OtherReadonlyNumber = 12;
}
```

Also, the readonly member can be made static.

Let's summarize:

- Const fields are assigned at compile time. Readonly fields can be assigned at runtime, in the constructor. The const field **must** be assigned at declaration. Readonly may be assigned at declaration or in the constructor.
- Consts can only be numbers, booleans, strings or a null reference, readonly values can be anything
- Consts can't be declared as static, because they are implicitly static. Readonly values can be static

Tip: other interview questions on this topic:

- "**Assume you need the PI number in your program. How would you define it?**"
  It is a bit tricky because the best answer is "I would not, I would use Math.PI that is already defined in C#". But if for some reason it would not be present, I would define it in some static class MathConsts as a const value.

- "**How can you prevent the field from being modified after it is set up in the constructor?**"
  By making it readonly.

- "**When do you use const, and when do you use readonly?**"
  I would use const when the value is known at the compile time, for example, mathematical constants like PI number should be const. I would use readonly when the value is known at runtime, but I do not wish it to change after it is assigned.

30. What is the difference between the "ref" and the "out" keywords?

Brief summary: ref passes the value type to a method by reference, which means any modifications of this value inside this method will be visible outside this method. out is a way of returning extra variables from a method.

First, let's focus on the **ref** keyword. Let's take a look at this simple method:

```
private static void AddOne(int a)
{
    Console.WriteLine($"Calling AddOne function");
    ++a;
}
```

This method looks like it increments the value it took as a parameter. Let's see its use in code:

```
var number = 10;
Console.WriteLine($"number is {number}");
AddOne(number);
Console.WriteLine($"number is still {number}");
```

Can you guess what will be printed to the console? If you are unsure, please consider going back to the "What is the difference between value types and reference types?" lecture.

Here is the output:

```
number is 10
Calling AddOne function
number is still 10
```

The number is not affected, because the integer is a value type. When passed to a method as a parameter, the copy of the initial variable is created. That's why after the AddOne method finishes execution, the number variable is still as it was.

But what if we want the number to be incremented? We would need to ensure it is somehow passed by reference to the AddOne method. And that's exactly the purpose of the **ref** keyword. It allows the modification of the value type passed to a method. Let's see it in code:

```
private static void AddOneByRef(ref int a)
{
    Console.WriteLine($"Calling AddOneByRef function");
    ++a;
}
```

Now, the following code…

```
var number = 10;
Console.WriteLine($"number is {number}");
AddOneByRef(ref number);
Console.WriteLine($"now number is {number}");
```

…will result in the following output:

```
Calling AddOneByRef function
now number is 11
```

The important restriction here is that the value passed by ref must be initialized before the method is called. For example, this will not compile:

```
int uninitializedValue;
AddOneByRef(ref uninitializedValue);
```

(local variable) int uninitializedValue

CS0165: Use of unassigned local variable 'uninitializedValue'

Moving on to the **out** keyword. Sometimes we need to return more than one result from the method. The classic example might be parsing a string to a number - we define a method taking in a string, and we would like to return an integer… but the parsing might not be successful. So we would also like to return a boolean value telling if the parsing was successful or not. And this is exactly the use for the out keyword:

```
private static bool TryParseToInt(string input, out int result)
{
    try
    {
        result = int.Parse(input);
        return true;
    }
    catch
    {
        result = 0;
        return false;
    }
}
```

Let's see how to use this method:

```
bool wasParsingSuccessful = TryParseToInt(validInput, out int result);
```

The boolean is returned as an ordinary output from a method, but the parsed value is returned by a special out parameter. This basically created a new variable that can then be used:

```
bool wasParsingSuccessful = TryParseToInt(validInput, out int result);
Console.WriteLine(result);
```

Thanks to the out keyword, we now have two variables returned from a method. In this example I declared the variable right at the place of the method call, but using an existing variable is also possible. In this case, we must skip the type at the place of the method call:

```
int variableForResult;
bool wasParsingSuccessful2 = TryParseToInt(validInput, out variableForResult);
```

The restriction of the out keyword is that the value must be assigned inside the method. For example, this will not compile:

```
private static bool TryParseToDouble(string input, out double result)
{
    try
    {
        result = double.Parse(input);
        return true;
    }
    catch
    {
        //this will not compile because result out parameter MUST be assigned a value in this function
        return false;
    }
}
```

If the parsing is not successful the catch clause will be executed. Inside, no assignment of the result variable exists, and it is obligatory for the out parameters.

The out parameter can be very useful, but it should not be overused. When tempted to use it, first consider if the method could not be split into two and if the results could be calculated separately. Also, when returning more than one extra result from a method, it may be a better idea to create a class or a struct to represent the result in one object.

Let's summarize:

- The purpose of the ref keyword is to allow the modification of the value type passed to a method. For example, when passing an int to a method with a ref keyword, it will allow us to modify its value as it was a reference type, and any changes made to this parameter will be visible after the method finishes.
- The purpose of the out keyword is to declare that the out parameter will be set inside the method. It **must** be set inside this method, even if it was initialized before.

The main differences are:

- The ref parameter must be initialized **before** being passed to the method. The out parameter may be but does not have to.
- The out parameter **must** be initialized **inside** the method body. The ref parameter may be but does not have to.

Tip: other interview questions on this topic:

- "**How to modify the value of value type inside the method, so this modification is visible after the method finishes?**"
  By passing it to the method as a ref parameter.
- "**How to return an additional piece of information from the method along with the return value?**"
  By using an out parameter. This is how it works with methods like "int.TryParse(string text, out bool wasParsingSuccessful)". Of course, there are

other ways to achieve it - for example, you can return a Tuple instead of a simple value.

31. What is the difference between an interface and an abstract class?

Brief summary: An interface defines what set of operations will be provided by any class implementing it - it does not provide any implementation on its own. An abstract class is like a general blueprint for derived classes. It may provide implementations of methods, contain fields, etc.

Before we delve into more technical details, let's try to understand the difference between interfaces and abstract classes at the conceptual level:

- An interface is an abstraction over **behavior**. It defines what an object can **do**. When you have a group of objects and they share similar behavior, they might have a common **interface**. For example, a bird, a kite, and a plane fly - so it makes sense for all of them to implement the IFlyable interface. When you are given an object implementing IFlyable interface, you might not be sure what that **is** - but you'll know it is **able** to fly. When you try to find what some objects have in common and a **verb** comes to mind, it means you probably want to use an interface.
- An abstract class is an abstraction over alikeness. It defines what an object **is**. When you have a group of objects, and they all belong to some general category of things, they might inherit from the same abstract class. For example, a bird, a snake and a dog all are animals - so it makes sense for them to inherit from abstract class Animal. When you are given an object that inherits from the Animal abstract class, you might not be sure how it **behaves** - but you know that it **is** some kind of animal. When you try to find what some objects have in common and a **noun** comes to mind, it means you probably want to use an abstract class.

Let's look more technically on what an interface and an abstract class are:

- An interface is a set of **definitions** of methods - it **does not** provide any implementation (at least in 99% of the cases - see the note about interfaces change in C# 8.0 at the end of this lecture). It specifies a **contract** that an implementing method will have to fulfill. When you implement an interface in your class, it means you declare that this class will provide all the methods from this interface. For example, one of the most commonly used interfaces in C# is ICollection - an interface that defines a set of methods related to working on collections - methods like Add, Contains, Remove, Clear, etc. ICollection only defines what methods a collection must provide, but they are not implemented in the interface itself. The concrete classes

implementing this interface - for example List - provide the implementations.

```
interface IFlyable
{
    void Fly(); //no "public", no method body
}

class Bird : IFlyable
{
    public void Fly()
    {
        Console.WriteLine("Flying using fuel of grain and worms.");
    }
}

class Drone : IFlyable
{
    public void Fly()
    {
        Console.WriteLine("Flying using energy stored in battery.");
    }
}
```

- An abstract class is a type that is too - *well* - abstract for the actual instances of it to exist. It represents some general category of things. It can have method implementations, but it can also contain abstract methods - methods with no bodies, that will have to be implemented in the inheriting classes. As in the example before, you can imagine an Animal abstract class. You can't have an object of type Animal - it's always some specific kind of animal, like a dog or a horse. Animal is just an abstraction over a whole category of creatures a bit similar to each other (and not similar at all to plants or fungi).

```
abstract class Animal
{
    public abstract void Move();
}

abstract class Mammal : Animal //abstract class inheriting from abstract class
{
    public void ProduceMilk()
    {
        Console.WriteLine("Producing milk to feed its young");
    }
}

class Snake : Animal
{
    public override void Move()
    {
        Console.WriteLine("Slithering on belly");
    }
}

class Dog : Mammal
{
    public override void Move()
    {
        Console.WriteLine("Running using four legs");
    }
}

class Cat : Mammal //does not compile - MUST provide implementation of the Move method
{
}
```

As you can see you can't create instances of an abstract class:

```
var animal = new Animal(); //can't create an instance of an abstract class
var mammal = new Mammal(); //can't create an instance of an abstract class

var dog = new Dog();
var snake = new Snake();
```

To summarize - the differences between an interface and abstract class are:

- Interface can't provide any implementation of the methods, an abstract class can (but doesn't have to if the method is abstract)
- All interface methods are by default public and they can't have any other access modifier specified. They can not be sealed or static (that wouldn't make sense because sealed or static methods can't be overridden). They also can't declare methods as abstract or virtual (that would be redundant because as methods with no bodies, meant to be implemented in classes implementing the interface, they are already kind of abstract and virtual).

- An interface can only contain methods or properties definitions - it can not have fields or constructors, while an abstract class can
- A class can implement multiple interfaces, but it can only inherit from one abstract class

	Interface	Abstract class
Abstraction over	behavior	alikeness
Defines what an object ...	...can do	...is
Group of objects share...	...behavior	...general category of things
Example	bird, kite and plane **can** fly	bird, snake and dog **are** animals
Not sure what it...	...is	...is able to do
Sure what it...	...is able to do	...is
Part of speech	verb	noun

Before we end this lecture I would like to mention one thing - **starting with C# 8.0 interfaces can have methods with bodies**. I decided to skip it in this course, as it exceeds junior level. If you are interested, please see https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/default-interface-methods-versions

Tip: other interview questions on this topic:

- "**Why can't you specify the accessibility modifier for a method defined in the interface?**"
  The point of creating an interface is to specify the public contract that a class implementing it will expose. Modifiers other than "public" would not make sense. Because of that, the "public" modifier is the default and we don't need to specify it explicitly.

- "**Why can't we create instances of abstract classes?**"
  Because an abstract class may have abstract methods which do not contain a body. What would happen if such a method was called? It doesn't have a body so the behavior would be undefined.

32. What is polymorphism?

Brief summary: Polymorphism is the provision of a single interface to entities of different types. In other words, there is a generic concept of something, and this concept can be made concrete by multiple types. All of them can be used wherever the generic concept is needed.

Polymorphism is the provision of a single interface to entities of different types. In other words, there is a **generic concept** of something, and this concept can be made **concrete** by multiple types. Still, all of them can be used wherever the generic concept is needed.

The word comes from Greek "**poly**" meaning "many" and "**morphe**" meaning "form" or "shape".

Let's see a simple code that will allow us to understand this cryptic definition better:

```
List<IFlyable> flyables = new List<IFlyable>
{
    new Duck(),
    new Plane(),
    new Kite()
};
```

First, we have a list of flying objects. We can fill this list with any objects implementing the IFlyable interface. In other words, the derived types can be used in place of the base type. The list doesn't care about the specific type of the object, as long as it implements the IFlyable interface.

Let's create a method using such a list:

```
private static void FlyAll(List<IFlyable> flyables)
{
    foreach (var flyable in flyables)
    {
        flyable.Fly();
    }
}
```

As you can see, this method is not aware of what concrete types are contained within the flyables list, nor does it care about that. It only needs to know that those objects provide the Fly method. It's the concrete classes - and only them - who are aware of the specific implementation. The specific types will be resolved at runtime, and only then the CLR determines which concrete method will be called on the objects from the flyables list.

This list can contain many objects of many different types - that should explain the Greek origin of the word "polymorphism". There is one list, but its elements can take many forms.

Polymorphism is one of the base concepts of object-oriented programming. We use it every time when we manipulate different objects implementing one interface, or inherit from the same base class. Polymorphism allows us to create a generic code that can be easily extended by simply providing new concrete types.

Tip: other interview questions on this topic:

- "**What mechanisms in C# allow us to use polymorphism?**"
  Interfaces, abstract classes, inheritance, virtual methods.

33. What's the difference between a virtual method and an abstract method?

Brief summary: A virtual method is a method that may be overridden in the derived class. An abstract method must be overridden (unless the derived class is abstract itself).

A virtual method is a method that might be overridden in inheriting class. In other words, the base class provides some implementation, but with making the method virtual the developer says "it's ok if you want to use your own implementation in the inheriting class".

Let's consider one of the most commonly seen examples: the ToString method from the System.Object class. All classes in C# inherit from the System.Object class, so they also inherit the default implementation of the ToString method - which by default returns the name of the type. But it is very common that we want to override this default behavior - imagine we have a User class, containing Name and Email - we would rather want ToString method to return something like "User name is {Name}, email is {Email}" instead of "SomeNamespace.User".

An abstract method is somewhat similar, but it does **not** provide any implementation in the base class. A non-abstract derived class **must** override the abstract method. Please note that every **abstract method is implicitly virtual**.

What's the difference between virtual and abstract methods?

- A virtual method has an implementation, and it gives the inheriting class an **option** to provide its own implementation
- An abstract method does not have an implementation. The non-abstract inheriting class **must** provide its own implementation

Let's see how virtual and abstract methods are used in practice. In the example below, the derived class **must** provide an implementation of the **abstract** method from the base class. As you can see, the base class does not provide any implementation.

```
public abstract class Printer
{
    public abstract void Print(string text);
}

public class TextFilePrinter : Printer
{
    public override void Print(string text)
    {
        File.WriteAllText("someFile.txt", text);
    }
}
```

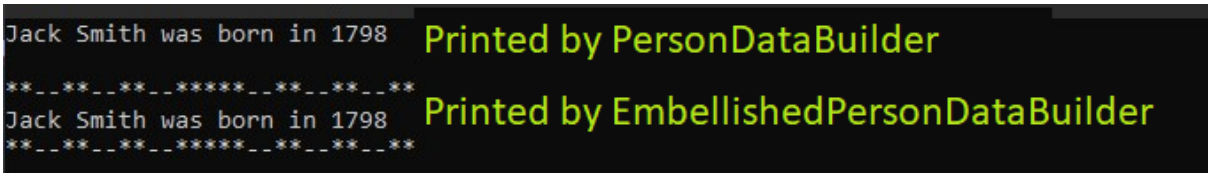
In the next example we can see how a virtual method from a base class is overridden in the derived class:

```
public class PersonDataBuilder
{
    public virtual string BuildPersonData(string name, string lastName, int yearOfBirth)
    {
        return $"{name} {lastName} was born in {yearOfBirth}";
    }
}

public class EmbellishedPersonDataBuilder : PersonDataBuilder
{
    public override string BuildPersonData(string name, string lastName, int yearOfBirth)
    {
        var prettyLine = "**--**--**--*****--**--**--**";
        return $"{prettyLine}\n" +
            $"{base.BuildPersonData(name, lastName, yearOfBirth)}\n" +
            $"{prettyLine}";
    }
}
```

Please note that virtual methods are executed in a **polymorphic** way - that means when the method is invoked the runtime environment checks the actual type of the object that this method is called upon, and uses the proper implementation. Let's consider the following code:

```
var personDataBuilders = new List<PersonDataBuilder>
{
    new PersonDataBuilder(),
    new EmbellishedPersonDataBuilder()
};
foreach(var personDataBuilder in personDataBuilders)
{
    Console.WriteLine(personDataBuilder.BuildPersonData("Jack", "Smith", 1798));
}
```

The foreach loop iterates over a collection of objects of PersonDataBuilder type. During the execution of the program, the runtime recognizes the second object in this collection as an EmbellishedPersonDataBuilder instance and calls the overridden method from this class. Because of that, the result of this program is:



When using virtual methods, remember that:

- the signature between the base class method and overridden derived class method must be the same
- the following modifiers cannot be used with the **override** modifier:
    - New - because method overriding and method hiding are mutually exclusive
    - Virtual - because the virtual modifier is explicitly inherited from the base class method
    - Static - because static methods cannot be virtual
- "Override" and "virtual" keywords can also be applied to properties, indexers, and events, not only methods

Virtual methods are one of the main tools used for achieving **polymorphism** in C#. (More on polymorphism can be found in "What is polymorphism?" lecture).

Tip: other interview questions on this topic:

- "**What is method overriding?**"
  It is providing a custom implementation of virtual or abstract methods in the child class.

- "**When a method must be overridden?**"
  In the non-abstract child class, if it was abstract in the base class.

- "**Are abstract methods virtual?**"
  Yes, all abstract methods are implicitly virtual.

34. What is the method overloading?

Brief summary: Method overloading is having a class with multiple methods with the same name, that differ only in parameters.

Method **overloading** is having a class with multiple methods with the same name that differ only in parameters.

Since the methods are named the same, the compiler must have another way to tell which one should be called. For example, they can be distinguished by a number of parameters.

Method **overloading** happens when we have multiple methods with the same name, that differ in:

The type of parameters:

```
private int Add(int a, int b)
{
    return a + b;
}

private string Add(string a, string b)
{
    return a + b;
}
```

The number of parameters:

```
private int Add(int a, int b)
{
    return a + b;
}

private int Add(int a, int b, int c)
{
    return a + b + c;
}
```

Order of parameters:

```
private void Add(List<int> list, int newElement)
{
    list.Add(newElement);
}

private void Add(int newElement, List<int> list)
{
    list.Add(newElement * 2);
}
```

Please note that we can't have multiple methods that differ only by return type:

```
private int Add(int a, int b)
{
    return a + b;
}

private string Add(int a, int b)
{
    return a.ToString() + b.ToString();
}
```

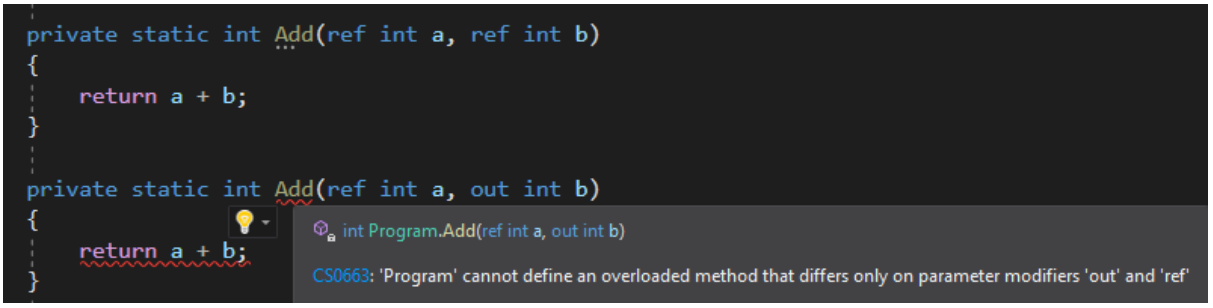
The reason for that is that the compiler would not know which method to call:

```
var someInt = Add(1, 2);
var someString = Add(1, 2); //how can the compiler know which one I mean?
```

It's ok to have methods with the same name when one is based on value parameters and the other on the ref and out parameters:

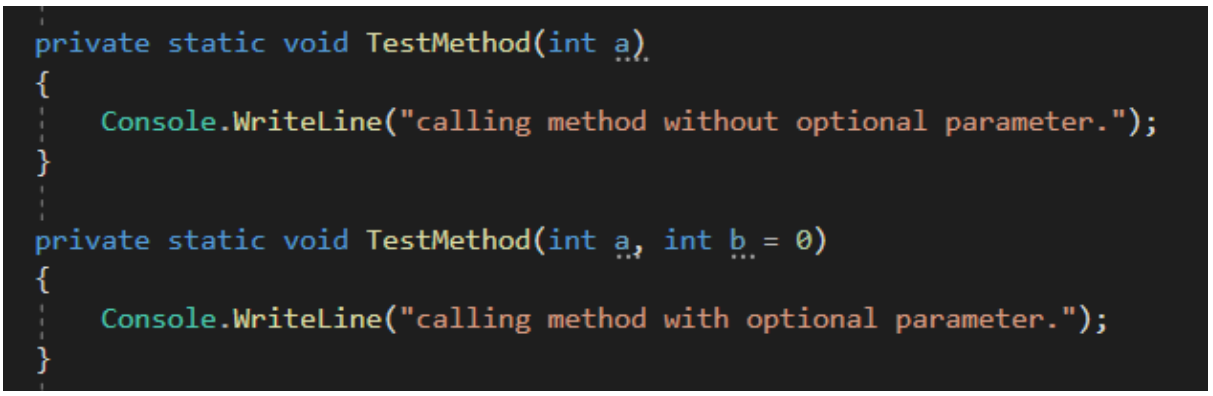
```
private static int Add(int a, int b)
{
    return a + b;
}

private static int Add(ref int a, ref int b)
{
    return a + b;
}
```

…but it is NOT ok to have two methods that only differ by ref/out modifiers:



The reason for that is pretty low-level - when C# is compiled to the Intermediate Language, it turns out that both methods will be compiled to the same signature, thus creating a conflict of methods in the CIL.

We can also overload methods with optional parameters:



The question here is what method will be called in such a scenario:

```
TestMethod(1);
```

The answer is: the method without the optional parameter. The compiler always chooses the method with the signature closest to what was called. In this case, we call a method with one parameter, so it chooses the method with exactly one parameter, not the one that may take one, but may also take two parameters.

Tip: other interview questions on this topic:

- "**If two methods have the same name and parameters but for one the last parameter is optional, which method will be used when all parameters are provided?**"
  The one without optional parameters has a priority.

- "**What's the difference between method overloading and method overriding?**"
  Method overloading is having a class with multiple methods with the same name that differ only in parameters. Method overriding is providing a custom implementation of virtual or abstract methods in the child class.

35. What is the difference between method overriding and method hiding?

Brief summary: Method overriding happens when the derived class provides its own implementation of a virtual or abstract method from a base class. Method hiding happens when there is a method in the derived class with the same name as the method in the base class, that does not override the base class method.

The difference between method overriding and method hiding is that:

- The method **overriding** happens when the **derived class provides its own implementation of a virtual** or abstract method from a base class. It is done with the "**override**" keyword.
- The method **hiding** happens when there is a method in the **derived class with the same name as the method in the base class**, that does not override the base class method. It is done with the "**new**" keyword.

To understand it better, let's see the following code:

```
public class Animal
{
    public string AsText()
    {
        return $"This is an animal of type: {GetDescription()}";
    }

    public virtual string GetDescription()
    {
        return "generic animal";
    }
}

public class Tiger : Animal
{
    public override string GetDescription()
    {
        return "tiger, the king of Asia";
    }
}

public class Lion : Animal
{
    public new string GetDescription()
    {
        return "lion, the ruler of Africa";
    }
}
```

The AsText method from the Animal class calls the virtual method GetDescription. The default implementation is provided in the Animal class (returning the "generic animal" string). We have two classes derived from Animal class - Tiger and Lion. Please note that in the Tiger class the GetDescription method is **overridden**, while in the Lion class it is **hidden** by using the new keyword. This is a very important difference.

Let's see those classes in use:

```
Animal genericAnimal = new Animal();
Console.WriteLine($"generic animal: {genericAnimal.AsText()}\n");

Animal tiger = new Tiger();
Console.WriteLine($"tiger: {tiger.AsText()}\n");

Animal lion = new Lion();
Console.WriteLine($"lion: {lion.AsText()}\n");
```

As you can see, every variable is of Animal base type. What do you think will be printed to the console? Take a second to try to figure it out.

You might be surprised what is printed for the lion:

```
generic animal: This is an animal of type: generic animal

tiger: This is an animal of type: tiger, the king of Asia

lion: This is an animal of type: generic animal
```

The lion did not print what the GetDescription method from the Lion class produces ("lion, the ruler of Africa"). It printed the "generic animal" description from the base class. Why is that?

The lion variable is of type Animal. When the AsText method is executed, it is done in a virtual way - that means, the runtime checks the actual type of this object and executes the overridden method from the child class. This is exactly what happens for the tiger, and it produces the "tiger, the king of Asia" string, as expected. Why doesn't the same thing happen for the lion? It is because the Lion class **does not override** the GetDescription method! It **hides** it - which means, it has a method "accidentally" named the same as the method in the base class, but except for that it has nothing to do with the base class method - it does not override it, so the runtime does not execute it in place of the GetDescription method from the base class. It uses the virtual method called GetDescription, and there is no such virtual method in the Lion class - that's why the method from the base class is used instead. This is exactly the same scenario as no GetDescription method would exist in the Lion class at all.

What would happen if the variables were not of type Animal, but of the derived types?

```
Tiger tiger2 = new Tiger();
Console.WriteLine($"tiger description: {tiger.GetDescription()}\n");
Console.WriteLine($"tiger2 description: {tiger2.GetDescription()}\n");

Lion lion2 = new Lion();
Console.WriteLine($"lion description: {lion.GetDescription()}\n");
Console.WriteLine($"lion2 description: {lion2.GetDescription()}\n");
```

Please note that this time we are executing the GetDescription methods, not the AsText methods. The result of such code would be:

```
tiger description: tiger, the king of Asia

tiger2 description: tiger, the king of Asia

lion description: generic animal

lion2 description: lion, the ruler of Africa
```

So in this case, the lion2, a variable of type Lion, finally produced the string "lion, the ruler of Africa". This is because we operate directly on the Lion type, and the runtime simply finds and executes the GetDescription method from the Lion class instead of doing it in a virtual way as before.

By now you might be thinking "Why use method hiding at all if it makes things so complicated?". Good point! It would be perfect not to have to use it at all. But imagine such a scenario: you develop an application that works on some geometric shapes. You know that there is no reason for "reinventing the wheel" so you reference some library that provides simple operations on shapes. Let's use Circle as an example:

```
public class Circle //imagine this is defined in an open source library that you use
{
    public double Radius { get; }
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Circumference()
    {
        return 2 * Math.PI * Radius;
    }
}
```

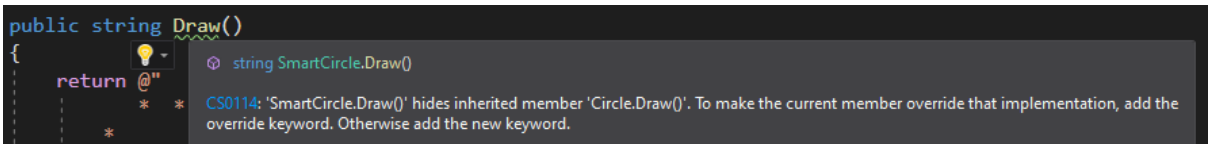
This class is perfect for your needs, it only misses a couple of things - you would like to be able to also calculate the area of a circle, as well as to be able to draw it in the console. So you decide to create your own class, that will derive from the Circle class but also provide the missing methods:

```
public class SmartCircle : Circle
{
    public SmartCircle(double radius) : base(radius)
    {
    }

    public double Area()
    {
        return Math.PI * Radius * Radius;
    }

    public string Draw()
    {
        return @"
              *  *
           *        *
          *          *
          *          *
           *        *
              *  *    ";
    }
}
```

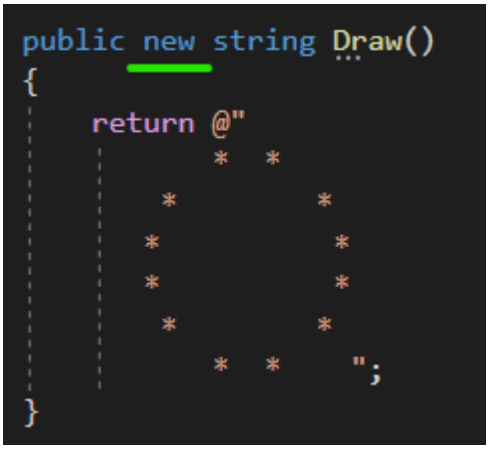
Great, now you can use the SmartCircle class and everything works fine. But then, after a couple of months, you update the version of the library that provided you with the Circle class. And you notice that two new methods were added:

```
public class Circle //imagine this is defined in an open source library that you use
{
    public double Radius { get; }
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Circumference()
    {
        return 2 * Math.PI * Radius;
    }

    public double Area()
    {
        return Math.PI * Radius * Radius;
    }

    public virtual string Draw()
    {
        return @"O";
    }
}
```

What a coincidence - the developers of the library added methods with the same signature as you have! While the Area method is exactly the same and you can safely remove your own from the SmartCircle class, the Draw method might be a problem - you don't really like the implementation provided by the library, you like your own better. This is where method hiding comes in handy - you can hide the base class method as kind of a declaration "I know there is a method with the same signature in the base class, but I want to use the method from the child class". Please note that in case of "collision" of method signatures in base and derived class, **method hiding is the default behavior** - you don't need to add the "new" keyword. Still, most of the IDE's will give you a warning:



This is a fair warning - it is better to add the new keyword explicitly, so other developers are sure this is intended behavior, not a coincidence. Let's fix the warning:



Of course, you might decide that you actually want to override the base class method instead of hiding it - in this case, you should use the override keyword. Another solution is to rename your own method to something else. This might sometimes be complicated to implement, especially if your project is big or other projects depend on it.

To summarize: method hiding lets us control what method is called when there is a conflict of method names in base and derived class. Method overriding is about providing a custom implementation in the derived class.

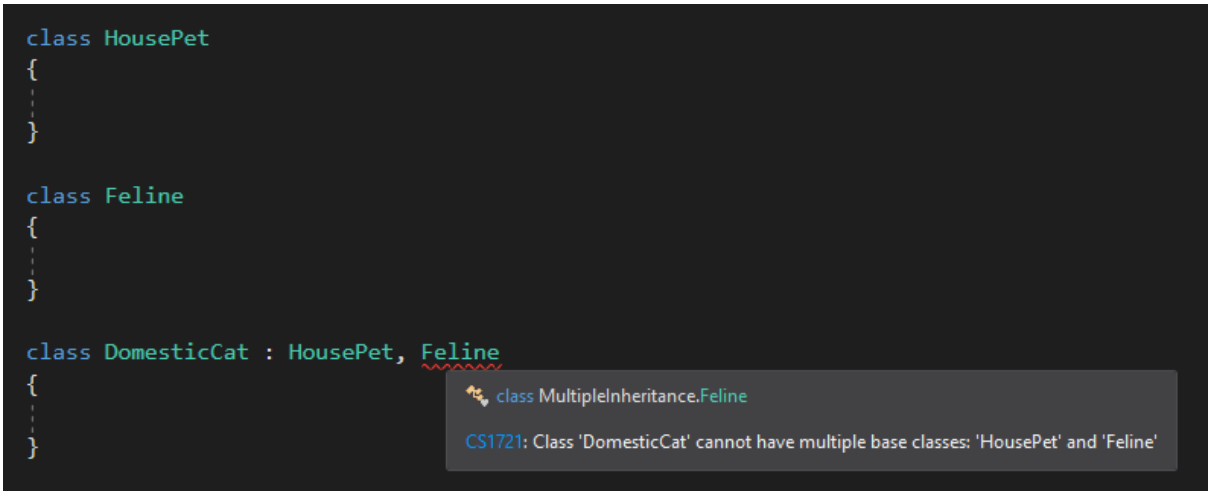
Tip: other interview questions on this topic:

- "**What keyword do you have to use to hide a base class method in the child class?**"
  Actually, I don't have to use any keyword - the method is hidden by default. I can do it explicitly with the "new" keyword.

36. Does C# support multiple inheritance?

Brief summary: No, C# does not support multiple inheritance. It does support implementing multiple interfaces, though.

Multiple inheritance takes place when one class is derived from more than one base class. It is **not** supported in C#. If it was, it would look somehow like this:

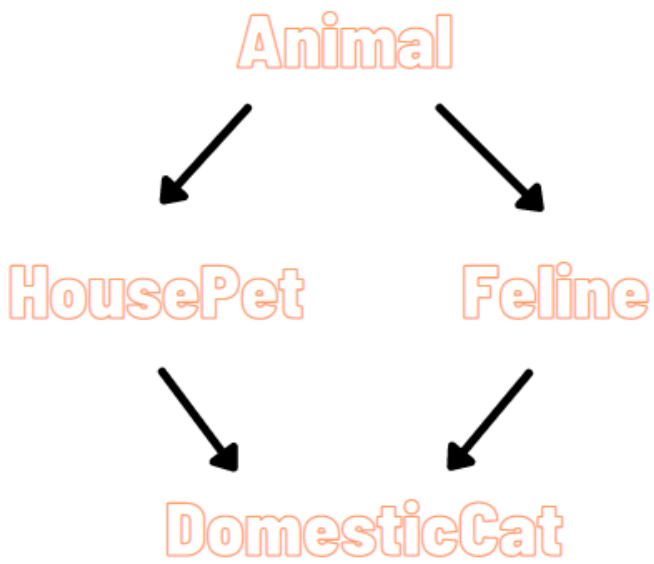


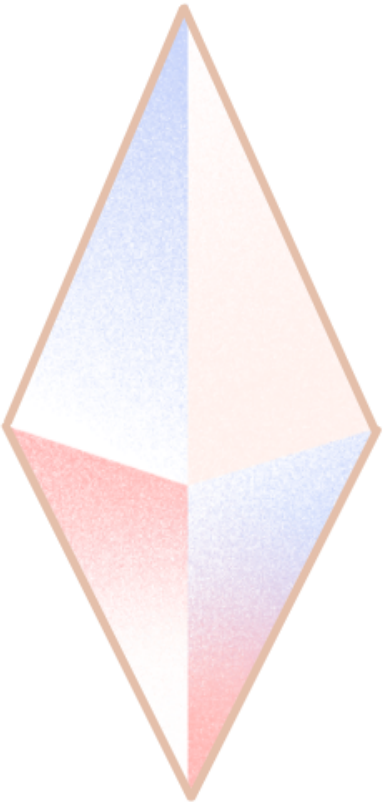
At first glance it may look like it makes sense. A domestic cat is a house pet, and it is a feline, so why not inherit from both of those classes? Actually, there are many languages that would allow us to do it, for example C++. Why did C# creators decide to not support this feature? This is probably because multiple inheritance leads to the so-called "**diamond problem**". To illustrate it, let's add another class to the picture:

```
abstract class Animal
{
    public abstract string MakeSound();
}

class HousePet : Animal
{
    public override string MakeSound()
    {
        return "<happy noises when human comes back home>";
    }
}

class Feline : Animal
{
    public override string MakeSound()
    {
        return "Purr purr, I'm a ball of fur";
    }
}

class DomesticCat : HousePet, Feline
{

}

class Program
{
    static void Main(string[] args)
    {
        var domesticCat = new DomesticCat();

        Console.WriteLine(domesticCat.MakeSound());
```

As you can see, both HousePet and Feline inherit from the Animal class, and they both override the MakeSound method. The DomesticCat class is derived both from Feline and HousePet, and it inherited the MakeSound method from both of them.

The inheritance diagram looks a bit like a diamond:





Now, when an instance of DomesticCat class is created and the MakeSound method is called, how can the CLR know which method to execute - the one from the HousePet class, or the one from the Feline class?

The diamond problem is mitigated in various ways in the languages supporting multiple inheritance, but for some people none of those solutions is perfect. The C#'s designers decided that multiple inheritance should simply not be part of the language. On the other hand, they decided that C# should support multiple interfaces implementation.

That makes more sense. It is less likely that one thing is two other things, but it is more natural that one thing behaves like two things. Remember, deriving from a class is like saying "this **is** the object of a base type", for example, Dog can inherit from Animal class because Dog **is** an Animal. On the other hand, implementing an interface is saying "this object can **behave** in the way described in the interface". For example, a Duck can fly, swim, walk and even dive (ducks are the best). That's why it feels natural that the Duck class would implement IFlyable, ISwimmable, IWalkable, and IDiveable interfaces.

Let's go back to the DomesticCat class example. As we said, deriving from two base classes is not allowed in C#. Let's change multiple inheritance to multiple interfaces implementation:

```
public interface IAnimal
{
    string MakeSound();
}

public interface IHousePet : IAnimal
{
}

public interface IFeline : IAnimal
{
}

public class HousePet : IHousePet
{
    public string MakeSound()
    {
        return "<happy noises when human comes back home>";
    }
}

public class Feline : IFeline
{
    public string MakeSound()
    {
        return "Purr purr, I'm a ball of fur";
    }
}

public class DomesticCat : IFeline, IHousePet
{
    public string MakeSound()
    {
        return "Purr purr, I'm a ball of fur, " +
            "but I am not too excited when human comes back home.";
    }
}
```

As you can see, now there is no dilemma if the method from one base class or the other should be used - because implementing the interface forced us to provide the implementation in the DomesticCat class.

Let's make sure the expected thing will be printed to the console:

```
DomesticCat domesticCat = new DomesticCat();
Console.WriteLine(domesticCat.MakeSound());
```

```
Purr purr, I'm a ball of fur, but I am not too excited when human comes back home.
```

Let's summarize. C# does **not** support multiple inheritance, as it leads to the **diamond problem**. Only **multiple interfaces implementation** is allowed.

Tip: other interview questions on this topic:

- "**What is the "diamond problem"?**"
  The diamond problem is an ambiguity that arises when class D is derived from classes B and C, which both derive from class A. When both classes B and C overrode a method from class A, then it is ambiguous which one would be used when this method is called on an object of class D.

37. What is the DRY principle?

Brief summary: "DRY" stands for "Don't Repeat Yourself" and it means that we shouldn't have multiple places where pieces of business knowledge are defined. Also, DRY is commonly considered a rule of avoiding code duplication.

"DRY" stands for "Don't Repeat Yourself". This principle says that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system" (according to "The Pragmatic Programmer" by Andrew Hunt and David Thomas, the book in which this principle was originally defined). To understand it we must first know what a "piece of knowledge is".

Piece of knowledge can be either:

- An algorithm
- A business functionality or policy defined for the application

Some of you might be a bit confused by now. "Wait, what? I thought DRY is about not copy-pasting the code!". Well, yes and no. The original principle from "The Pragmatic Programmer" says nothing about the code. On the other hand, of course you should not be copy-pasting the code. To make things easier, let's split the DRY rule into two pieces:

- The original DRY principle that says the **pieces of knowledge** should not be duplicated within the application
- The common-sense, but not the original DRY principle that says you should **not duplicate the code**

There is an overlap between those pieces, of course - you might be duplicating business knowledge by duplicating the code, but you don't have to. Let's see an example of a "piece of knowledge". You are developing an application for an e-commerce system like Ebay or Amazon. You are informed by the business analyst that the goods bought via the platform can be returned within 30 days. **This is a piece of knowledge**. Let's see the code:

```
class OnlineStore
{
    public DateTime ReturnDateDeadline(DateTime purchaseDate)
    {
        return purchaseDate.AddDays(30);
    }

    public bool IsAfterPossibleReturnDate(DateTime purchaseDate)
    {
        return (DateTime.Now - purchaseDate).TotalDays > 30;
    }
}
```

We have two methods here - first calculates the deadline for the return of the goods, and the other checks if we are past such a deadline. **No code is duplicated** here - those methods differ significantly, and obviously no one copy-pasted any piece of code. On the other hand, **a piece of knowledge is repeated** - we have **two** places where we define the 30-days returns policy. That breaks the original DRY principle. This is of course not good - if the business policy changes, we will have two places to fix. Not only is that laborious, but it is also error-prone - what if we miss a place in the application where this policy was defined?

Let's refactor it before we move on:

```
class OnlineStore
{
    public const int DaysForReturn = 30;

    public DateTime ReturnDateDeadline(DateTime purchaseDate)
    {
        return purchaseDate.AddDays(DaysForReturn);
    }

    public bool IsAfterPossibleReturnDate(DateTime purchaseDate)
    {
        return ReturnDateDeadline(purchaseDate) < DateTime.Now;
    }
}
```

Great. Now the 30-days return policy is defined in one place only.

The above code was an example of a situation when a piece of knowledge is repeated, but the code isn't. Let's see the opposite - a piece of knowledge will be defined once, but the code will be duplicated:

```
public void CommitOrder(Order order)
{
    if(string.IsNullOrEmpty(order.CustomerId))
    {
        throw new Exception($"The CustomerId must not be empty");
    }
    if (string.IsNullOrEmpty(order.ProductId))
    {
        throw new Exception($"The ProductId must not be empty");
    }

    //saving to database here...
    Console.WriteLine("Order committed and saved to database");
}
```

Let's see - the piece of knowledge that is implemented here is "When committing an order, the ID of a customer and the ID of the product must be provided and non-empty". This piece of knowledge is defined only once, so the DRY principle is not broken here. On the other hand, the code itself is duplicated - and it shouldn't be. Code duplication is not a good thing - for example, if the exception message changes, we will have two places to fix. Let's refactor this code:

```
public void CommitOrder(Order order)
{
    Validate(order.CustomerId, nameof(order.CustomerId));
    Validate(order.ProductId, nameof(order.ProductId));

    //saving to database here...
    Console.WriteLine("Order committed and saved to database");
}

private void Validate(string idToBeValidated, string propertyName)
{
    if (string.IsNullOrEmpty(idToBeValidated))
    {
        throw new Exception($"The {propertyName} must not be empty");
    }
}
```

Great - now we got rid of the code duplication.

The original DRY principle should never be broken - it's always the problem if a piece of business knowledge is duplicated in the application. But can the same be said for **code duplication**? Actually, no. It is **almost** always a good idea to avoid code duplication. Almost.

Let's see an example when this is not such a great idea to avoid code duplication. Back to the example of an e-commerce platform. If you remember, first we defined a policy that a customer has 30 days to return bought goods. Now imagine you are implementing a **new** feature, and the business analyst told you that after the goods are returned the customer service has 30 days to **refund** money to the customer. Then you are asked to implement methods that check if it is after the refund date and to calculate the deadline for the refund. The methods that handle this would be:

```
public const int DaysForRefund = 30;

public DateTime RefundDateDeadline(DateTime purchaseDate)
{
    return purchaseDate.AddDays(DaysForReturn);
}

public bool IsAfterPossibleRefundDate(DateTime purchaseDate)
{
    return RefundDateDeadline(purchaseDate) < DateTime.Now;
}
```

The code that handles **refunds deadline** is copy-pasted from the code that was handling the **deadline of the return**. You know you shouldn't have done it, and that good programmers never copy-paste the code. You fix your mistake quickly:

```
public DateTime RefundDateDeadline(DateTime purchaseDate)
{
    return ReturnDateDeadline(purchaseDate);
}

public bool IsAfterPossibleRefundDate(DateTime purchaseDate)
{
    return IsAfterPossibleReturnDate(purchaseDate);
}
```

Smart. You reused the methods that you already have, and this is a good thing, right? Good programmers always reuse the code that is already implemented. The code works excellent and you are happy with yourself.

Not so quickly.

This is actually **not** good. You merged two pieces of functionality into one - calculating the return deadline and calculating the refund deadline. They may work the same way (for now) but they are actually not related from a business perspective. They are the same by coincidence, not for business-based reasons. There is a big chance that one will change while the other will not (for example, the deadline of the return will be extended to 100 days). Then, you would have to split this code back into two pieces. In huge, real-life applications it might be very hard. Let's make sure that in our tiny project it is done correctly:

```
//the below are the same by coincidence - we should NOT have one constant for it
public const int DaysForReturn = 30;
public const int DaysForRefund = 30;

public DateTime ReturnDateDeadline(DateTime purchaseDate)
{
    return purchaseDate.AddDays(DaysForReturn);
}

public bool IsAfterPossibleReturnDate(DateTime purchaseDate)
{
    return IsBeforeNow(ReturnDateDeadline(purchaseDate));
}

public DateTime RefundDateDeadline(DateTime purchaseDate)
{
    return purchaseDate.AddDays(DaysForRefund);
}

public bool IsAfterPossibleRefundDate(DateTime purchaseDate)
{
    return IsBeforeNow(RefundDateDeadline(purchaseDate));
}

private bool IsBeforeNow(DateTime dateTime)
{
    return dateTime < DateTime.Now;
}
```

The code looks better now. We have a little code duplication, but at least the things that should be kept separate from a business perspective are indeed separated.

Another situation when avoiding code duplication at all cost is not such a good idea, is when it leads to a very complex and unmanageable abstraction. Sometimes it's better to have some code duplication at the beginning of work, and then see how the code evolves, instead of building a very abstract code from the scratch, that will in the end be thrown away because the requirements changed so much.

As a final note, I want to make something clear: in most cases, code duplication **is a bad thing** and should be avoided, so please don't feel encouraged by me to copy-paste the code. Just keep in mind that avoiding code duplication at all costs may lead to some nasty problems in the application design, and it shouldn't be done blindly.

I don't want to get into further details on this topic, because it could be a subject of an hour-long lecture. If you are interested I recommend you this post by Sandi Metz, who also touches a very interesting problem of "wrong abstraction" - the situation when abstraction is introduced to avoid code duplication, but it actually leads to high coupling and unmanageable code: https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

Tip: other interview questions on this topic:

- "**What are the use cases when having code duplication is reasonable?**"
  First, when two pieces of code are identical but handle different business cases, especially when there is a chance that one business case will change independently from the other. Second, when the price to pay for avoiding code duplication is to create a very complex and unmanageable abstraction.

38. What is the "magic number" antipattern?

Brief summary: A magic number is an unnamed hard-coded value used directly in the code.

A **magic number** is an unnamed hard-coded value used directly in the code, without any explanation why this particular value is used. In a broader meaning, also a string constant or any other simple type constant can be considered a magic "number". A "magic number" is a value that should be given a symbolic name, but was instead used in the code as a literal, often in more than one place.

Let's consider this piece of code:

```
class PriceCalculator
{
    public double Calculate(
        double basePrice, bool isFrozen, bool isBakedGoods,
        bool isPie, int daysToExpiry)
    {
        if (isFrozen)
        {
            return basePrice * 1.05;
        }
        if (isBakedGoods && daysToExpiry <= 14)
        {
            return basePrice * 1.05;
        }
        if (isPie && daysToExpiry <= 45)
        {
            return basePrice * 1.08;
        }
        if (isPie && daysToExpiry > 45)
        {
            return basePrice * 1.23;
        }
        else
        {
            throw new Exception("Invalid product");
        }
    }
}
```

At first, you might be confused: what are all those numbers? What are 1.05, 1.08, and 1.23, and what are 14 and 45? You might try to guess from the context: they might be some kind of price multipliers - probably taxes (?), and the others might be some kind of expiry days limits… probably.

Here is a general rule - when reading the code, the developer shouldn't be forced to guess anything.

To satisfy your curiosity about the above code: the 1.05, 1.08, and 1.23 are VAT rates in Poland (5%, 8%, and 23%). 14 and 45 are expiry time limits in days, used to determine which VAT rate should be used for specific baked goods. Yes, this algorithm is complicated, especially given that I implemented the calculation ***only*** *for the baked*

goods. It seems that sometimes it is easier to be a software developer than an owner of a baking business.

Another problem with using magic numbers is that usually they are used in more than one place. Even in this short code 1.05 and 45 are used twice. In a real-life application there could be thousands of places where VAT rates are used. Now, let's imagine a miracle happened and VAT rates were lowered in Poland - it would be a nightmare to fix them in the whole application. That's why **using magic numbers is considered an antipattern and a code smell.**

Let's fix this code. First, let's create a dedicated class that holds all the constants related to tax law. In case anything changes in the tax law, we will only have one place to change in the whole project:

```
class TaxRulesProvider
{
    public const double VatRateLow = 1.05;
    public const double VatRateMedium = 1.08;
    public const double VatRateHigh = 1.23;

    public class BakingIndustry
    {
        public const int ShortExpiryLimit = 14;
        public const int LongExpiryLimit = 45;
    }
}
```

Now, let's use those constants in the calculation:

```
class PriceCalculator
{
    public decimal Calculate(
        decimal basePrice, bool isFrozen, bool isBakedGoods,
        bool isPie, int daysToExpiry)
    {
        if (isFrozen)
        {
            return basePrice * TaxRulesProvider.VatRateLow;
        }
        if (isBakedGoods && daysToExpiry <=
            TaxRulesProvider.BakingIndustry.ShortExpiryLimit)
        {
            return basePrice * TaxRulesProvider.VatRateLow;
        }
        if (isPie && daysToExpiry <=
            TaxRulesProvider.BakingIndustry.LongExpiryLimit)
        {
            return basePrice * TaxRulesProvider.VatRateMedium;
        }
        if (isPie && daysToExpiry >
            TaxRulesProvider.BakingIndustry.LongExpiryLimit)
        {
            return basePrice * TaxRulesProvider.VatRateHigh;
        }
        else
        {
            throw new Exception("Invalid product");
        }
    }
}
```

The code is much more readable now. The developer who will read it will not need to guess anything. Also, any changes to the constants will be easy to make.

Tip: other interview questions on this topic:

- "**What is a code smell**?"
  A code smell is a characteristic of the code that indicates some deeper problem. Code smells could be magic numbers, code duplications, large classes, etc.

- "**What's the alternative for defining constant values as consts in the source code? Where else can they be defined?**"

We could define them in some kind of configuration file. It has the advantage over using consts, because changing them can be done by a non-programmer and it doesn't require recompilation and redeployment of the code.

39. Why is using the "goto" keyword considered a bad practice?

Brief summary: The goto statement transfers the program execution directly to a labeled statement. Using it is widely considered a bad practice, as it increases the complexity of the code. The flow of the program is tricky to follow, making reading and debugging the code harder. Also, it can lead to the unintentional creation of infinite loops.

The **goto** statement transfers the program execution directly to a labeled statement. Using it is widely considered a bad practice, as it increases the complexity of the code. The flow of the program is tricky to follow, making reading and debugging the code harder. Also, it can lead to the unintentional creation of infinite loops.

The goto statement transfers the program execution directly to a labeled statement. Let's see the simple example:

```
public int? TransformNumberToNullWhenZero(int a)
{
    if(a == 0)
    {
        goto HandleZeroCase;
    }

    return a;

    HandleZeroCase:
    return null;
}
```

This method returns the unchanged number that was provided as a parameter unless the number is zero - then it returns null. The "goto HandleZeroCase" command simply transfers the program flow to the "HandleZeroCase:" point, in this case, bypassing the "return a;" statement. Of course, this could be done without goto, but I used it for presentation purposes.

Using goto is widely considered a bad practice. My aim in this lecture is **not** to make a point if it is a bad practice or it isn't - I'll simply try to explain why so many people from the programming community **consider** it a bad practice, so you can be prepared if such a question pops up during the technical interview. I will also try to show you some advantages of using the goto statement.

We know now that goto is used to "jump" to a specific place in the code. One might think that this is pretty convenient. Unfortunately, in programming, much as in real life, simple and easy solutions are not always the best ones. Let's consider the main downsides of goto:

- **Complexity -** functions using goto are harder to understand
    - Goto might surprisingly redirect the execution of the program to some completely different area of the code
    - Goto might create "hidden" loops in the code - that means, loops that are not created with dedicated keywords like "for", "foreach", "while" or "do while".
    - Goto might render some code unreachable, but it will not be obvious to the person reading the code
- **Harder debugging** - debugging is one of the most powerful tools programmers use. It allows us to follow the execution of the code step-by-step, which is critical when it comes to solving bugs or understanding the flow of the code. When we use goto, the flow of the program becomes much more complicated - the execution jumps from one place to another, often under some conditions, and following it with the debugger might leave us more confused than we were before debugging.
- **Infinite loops** - when using goto, we might accidentally create an infinite loop in the code. The below code is just a simple example and it may seem like something easy to avoid, but in real-life, in much more complex code full of conditional operations, there is a real risk of accidentally creating such an infinite loop.
-

```
class GotoInfiniteLoop
{
    public void InfiniteLoop()
    {
        GotoMarker:
        goto GotoMarker;
    }
}
```

Let's see some code that heavily relies on goto. I'll intentionally not explain what this code does. Try to read it and figure it out. Later we will refactor this code to

not use the goto statement and hopefully, it will become obvious what this code does.

```
class VeryUglyPersonalDataFormatter
{
    private readonly IDatabase _database;

    public VeryUglyPersonalDataFormatter(IDatabase database)
    {
        _database = database;
    }

    public string FormatPersonalData(string personId)
    {
        var person = _database.GetPerson(personId);
        if (person == null)
        {
            goto HandleError;
        }

        var petId = person.PetId;
        ReadPet:
        var personsPet = _database.GetPet(petId);
        if (personsPet == null)
        {
            if (person.FamilyPetId != null && person.PetId != person.FamilyPetId)
            {
                petId = person.FamilyPetId;
                goto ReadPet;
            }
            goto HandleError;
        }


        if (person.DateOfBirth == null)
        {
            Console.WriteLine("The date of birth is not known");
            goto FormatPersonWithUnknownDateOfBirthText;
        }
        else
        {
            goto FormatPersonText;
        }


        FormatPersonText:
        return $"{person.Name} {person.LastName} with pet " +
            $"{(personsPet.PetId == null ? "unknown" : personsPet.PetId)}" +
            $" born in {person.DateOfBirth.Value.Year}";

        FormatPersonWithUnknownDateOfBirthText:
        return $"{person.Name} {person.LastName} with pet " +
            $"{(personsPet.PetId == null ? "unknown" : personsPet.PetId)}" +
            $" (date of birth is unknown)";

        HandleError:
        Console.WriteLine("Database read error occured");
        return null;
    }
}
```

Did you manage to figure out what this code does? If not, I am not surprised. Let's see the refactored version of this code, which does not use goto at all. We shall see if it is easier to understand:

```
class PersonalDataFormatter
{
    private readonly IDatabase _database;

    public PersonalDataFormatter(IDatabase database)
    {
        _database = database;
    }

    public string FormatPersonalData(string personId)
    {
        var person = _database.GetPerson(personId);
        var personsPet = GetPet(person);
        if (person == null || personsPet == null)
        {
            Console.WriteLine("Database read error occured");
            return null;
        }

        return person.DateOfBirth == null ?
            FormatPersonWithUnknownDateOfBirth(person, personsPet) :
            FormatPerson(person, personsPet);
    }

    private Pet GetPet(Person person)
    {
        return person == null ? null : _database.GetPet(person.PetId ?? person.FamilyPetId);
    }

    private static string FormatPerson(Person person, Pet personsPet)
    {
        return FormatPerson(person, personsPet, $"born in {person.DateOfBirth.Value.Year}");
    }

    private static string FormatPersonWithUnknownDateOfBirth(Person person, Pet personsPet)
    {
        Console.WriteLine("The date of birth is not known");
        return FormatPerson(person, personsPet, $"(date of birth is unknown)");
    }

    private static string FormatPerson(Person person, Pet personsPet, string dateOfBirthInformation)
    {
        return $"{person.Name} {person.LastName} with pet " +
               $"{(personsPet.PetId == null ? "unknown" : personsPet.PetId)}" +
               $" {dateOfBirthInformation}";
    }
}
```

Much better now. The program flow no longer jumps from one goto to another. Now it is clear what is going on - we try to read a person's information from the database, as well as a person's pet's (using the FamilyPetId if PetId is null). Then we return the personal data as a string, formatting it slightly differently if the date of birth is null.

In general, in most cases, it is better not to use goto than to do it. If you are tempted to use goto, please take a moment to consider if a better design is possible.

Nonetheless, there are actually some cases when using goto is considered acceptable. Please be careful with those scenarios though - always consider if the same effect cannot be achieved without goto.

Let's list the most common examples of acceptable usage of goto, and we will dive into details next:

- Breaking out of nested loops
- Common cleanup logic
- Performance optimization

The first case is **breaking out of nested loops**. Imagine three loops nested in one another. Let's say we want to break if some limit of operations is reached.

```
public int NestedLoops(int [][][] input)
{
    int totalNumberOfCalculations = 0;
    int maxNumberOfCalculations = 500;
    for (int i = 0; i < input.Length; i++)
    {
        for (int j = 0; j < input.Length; j++)
        {
            for (int k = 0; k < input.Length; k++)
            {
                //do some calculations here...
                ++totalNumberOfCalculations;
                if (totalNumberOfCalculations == maxNumberOfCalculations)
                {
                    break;   //unfortunately, this will only break from the inner-most loop
                }
            }
        }
    }

    return totalNumberOfCalculations;
}
```

The problem in the above code is that the "break" will only break from the inner-most loop. What if we want to break from all of them? Using goto might be the simplest solution:

```
public int NestedLoops(int [][][] input)
{
    int totalNumberOfCalculations = 0;
    int maxNumberOfCalculations = 500;
    for (int i = 0; i < input.Length; i++)
    {
        for (int j = 0; j < input.Length; j++)
        {
            for (int k = 0; k < input.Length; k++)
            {
                //do some calculations here...
                ++totalNumberOfCalculations;
                if (totalNumberOfCalculations == maxNumberOfCalculations)
                {
                    goto AfterLoop;
                }
            }
        }
    }
    AfterLoop:
    return totalNumberOfCalculations;
}
```

This way, we will leave all three loops and go straight to the "AfterLoop" marker. In most cases such code can easily be refactored by using "return" in the loop we want to break:

```
public int NestedLoops(int [][][] input)
{
    int totalNumberOfCalculations = 0;
    int maxNumberOfCalculations = 500;
    for (int i = 0; i < input.Length; i++)
    {
        for (int j = 0; j < input.Length; j++)
        {
            for (int k = 0; k < input.Length; k++)
            {
                //do some calculations here...
                ++totalNumberOfCalculations;
                if (totalNumberOfCalculations == maxNumberOfCalculations)
                {
                    return totalNumberOfCalculations;
                }
            }
        }
    }
    return totalNumberOfCalculations;
}
```

There are scenarios when using the return is not that simple - and then using goto might be considered a better choice.

Another scenario is **common cleanup logic**. Consider the following code:

```
public bool CheckIfAllIdsExistInTheDatabase(List<string> ids, EntityType entityType)
{
    foreach (var id in ids)
    {
        switch (entityType)
        {
            case EntityType.Person:
                var person = _database.GetPerson(id);
                if (person == null)
                {
                    goto ReportError;
                }
                break;
            case EntityType.Pet:
                var pet = _database.GetPet(id);
                if (pet == null)
                {
                    goto ReportError;
                }
                break;
            default:
                goto ReportError;
        }
    }

    return true;

    ReportError:
    _logger.LogError("Database read error");
    return false;
}
```

Again, this could be refactored to not to use goto:

```
public bool CheckIfAllIdsExistInTheDatabase(List<string> ids, EntityType entityType)
{
    foreach (var id in ids)
    {
        switch (entityType)
        {
            case EntityType.Person:
                var person = _database.GetPerson(id);
                if (person == null)
                {
                    LogError();
                    return false;
                }
                break;
            case EntityType.Pet:
                var pet = _database.GetPet(id);
                if (pet == null)
                {
                    LogError();
                    return false;
                }
                break;
            default:
                LogError();
                return false;
        }
    }

    return true;
}

private void LogError()
{
    _logger.LogError("Database read error");
}
```

Which one is better is a matter of personal taste. For example, I personally never use goto, but there are people who find it a lesser of two evils in some scenarios.

The third case of goto being acceptable is when making some low-level performance improvements. The Goto statement executes very quickly and it may give some methods a performance boost. On the other hand, usually the compiler is not able to optimize code using gotos. I will not delve into details of this use case of goto, as this is way beyond junior level.

Tip: other interview questions on this topic:

- "**How to simply break out from deeply-nested loops?**"
  For example, by using goto. We can also use the return keyword, but sometimes it requires moving the loop to a separate method.

- "**What are the use cases when using goto might be a good idea?**"
  Breaking out from nested loops, common cleanup logic, and performance optimizations.

40. What is the "spaghetti code"?

Brief summary: "Spaghetti code" is a pejorative term used to describe code that is messy, tangled, and hard to maintain.

"**Spaghetti code**" is a pejorative term used to describe code that is messy, tangled, and hard to maintain. You can imagine such code as a bowl of spaghetti, with all pasta threads being entangled together, making it very hard to say where one starts and where the other ends. It would be hard to follow a single thread without moving the others - and it is as hard to make any change in the spaghetti code, without affecting many other pieces of code.

There are plenty of issues that make code a "spaghetti code", so let's just list a couple of them:

- Huge classes and methods
- Multiple responsibilities in one class (see the "What is the "S" in the SOLID principles?" lecture)
- Using the "goto" keyword (see the "Why is using the "goto" keyword considered a bad practice?" lecture)
- Bad naming
- Magic numbers (see the "What is the "magic number" antipattern?" lecture)
- Code repetition (see the "What is the DRY principle?" lecture)

Let's see some spaghetti code, and then let's try to refactor it. Imagine you were told to implement such an interface:

```
interface IQuadraticFunctionRootsCalculator
{
    void Calculate();
}
```

The class implementing such an interface should calculate the roots of a quadratic function that is defined by a user via console interaction. This interface is not very good - since the Calculate method returns void it forces the implementer to mix reading from the console with actual roots calculation - but let's just leave it as it is for the purpose of the exercise.

Just a quick reminder: a quadratic function looks like this:

$$f(x) = ax^2 + bx + c$$

This is how you calculate the roots of a quadratic function:

- If discriminant (delta) > 0 we have two roots:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

- If discriminant (delta) = 0 we have one root:

$$x = \frac{-b}{2a}$$

- If discriminant (delta) < 0 we have no roots.

Now, let's see spaghetti code that actually implements the interface we defined before. If I were you, I would check out the solution linked to this course to see this code in the IDE.

Brace yourself, as this is going to be ugly:

```
class SpaghettiQuadraticFunctionRootsCalculator : IQuadraticFunctionRootsCalculator
{
    public void Calculate()
    {
        var f = false; //means "is finished"
        while (!f)
        {
            Console.WriteLine("Quadratic Function: y = ax^2 + bx + c");

            //variables for a,b,c
            double a;
            string astr;
            double b;
            string bstr;
            double c;
            string cstr;
            do
            {
                //reading a
                Console.WriteLine("Enter a");
                astr = Console.ReadLine();
                if (!double.TryParse(astr, out a))
                {
                    Console.WriteLine("Invalid format, please try again.");
                }
            }
            while (!double.TryParse(astr, out a));

            do
            {
                //reading b
                Console.WriteLine("Enter b");
                bstr = Console.ReadLine();
                if (!double.TryParse(bstr, out b))
                {
                    Console.WriteLine("Invalid format, please try again.");
                }
            }
            while (!double.TryParse(bstr, out b));
```

```
        do
        {
            //reading c
            Console.WriteLine("Enter c");
            cstr = Console.ReadLine();
            if (!double.TryParse(cstr, out c))
            {
                Console.WriteLine("Invalid format, please try again.");
            }
        }
        while (!double.TryParse(cstr, out c));

        var d = b * b - 4 * a * c;
        if (d > 0)
        {
            Console.WriteLine("Two roots:");
            Console.WriteLine((-b - Math.Sqrt(d)) / (2 * a));
            Console.WriteLine((-b + Math.Sqrt(d)) / (2 * a));
        }
        else
        {
            if (d == 0)
            {
                Console.WriteLine("One root:");
                Console.WriteLine((-b + Math.Sqrt(d)) / (2 * a));
            }
            else Console.WriteLine("Zero roots.");
        }
```

```
            bool ok;
            string b1str;
            Console.WriteLine("Run calculation again? Enter Y or N");
            do
            {
                b1str = Console.ReadLine();
                if (b1str == "Y")
                {
                    f = false;
                    ok = true;
                }
                else if (b1str == "N")
                {
                    f = true;
                    ok = true;
                }
                else {
                Console.WriteLine("Invalid format, please try again. Enter Y or N");
                   ok = false; }

            }
            while (!ok);

            }
} }
```

If you read this code whole and you understood what is going on - I'm impressed. There are plenty of things there that are just terrible. Not only is this code hard to read and understand, but it is also a breeding ground for bugs. Also, if we needed to make any changes in this code, it would be laborious and error-prone.

Let me point out some of the most outstanding problems in this code:

- One class doing everything
- Bad naming + unnecessary comments. Instead of this:

  ```
  var f = false; //means "is finished"
  ```

- …we should simply rename this variable to "isFinished" and remove the comment
- Code repetitions. This code:

```
do
{
    //reading a
    Console.WriteLine("Enter a");
    astr = Console.ReadLine();
    if (!double.TryParse(astr, out a))
    {
        Console.WriteLine("Invalid format, please try again.");
    }
}
while (!double.TryParse(astr, out a));
```

- …is repeated three times for each of a, b, and c variables
- More unnecessary comments:

```
//reading a
Console.WriteLine("Enter a");
```

- …since the code itself says that we are reading "a" here. No need to add a comment.
- Inconsistent/bad formatting, like in those parts of code:

```
                else { Console.WriteLine("Invalid format, please try again. Enter Y or N"); ok = false; }
            }
            while (!ok);

            }
} }
```

- Redundant calculations, like trying to parse string to double twice:

```
do
{
    //reading a
    Console.WriteLine("Enter a");
    astr = Console.ReadLine();
    if (!double.TryParse(astr, out a))
    {
        Console.WriteLine("Invalid format, please try again.");
    }
}
while (!double.TryParse(astr, out a));
```

Let me create an alternative implementation of this interface, this time in a better style. First, let's take refactor the main class:

```
class QuadraticFunctionRootsCalculator : IQuadraticFunctionRootsCalculator
{
    public void Calculate()
    {
        var isFinished = false;
        while (!isFinished)
        {
            Console.WriteLine("Quadratic Function: y = ax^2 + bx + c");

            double a = ConsoleReader.ReadDouble("a");
            double b = ConsoleReader.ReadDouble("b");
            double c = ConsoleReader.ReadDouble("c");

            var roots = MathUtilities.CalculateQuadraticFunctionRoots(a, b, c);
            if (roots.AreTwo)
            {
                Console.WriteLine($"Two roots: {roots.FirstRoot}, {roots.SecondRoot}");
            }
            else if (roots.IsOne)
            {
                Console.WriteLine($"One root: {roots.FirstRoot}");
            }
            else
            {
                Console.WriteLine("Zero roots.");
            }

            isFinished = ConsoleReader.ReadBool("Run calculation again?");
        }
    }
}
```

As you can see this class is much smaller now. I've moved the operations of reading from the console to a separate class ConsoleReader. I've also created a MathUtilities class that only deals with the actual calculation of the square function roots. Since this calculation produces a result that may have zero, one, or two numbers within, I also created a special class QuadraticFunctionRoots to represent such a result.

Let's see those classes. First, the ConsoleReader:

```
static class ConsoleReader
{
    public static double ReadDouble(string variableName)
    {
        double result;
        bool wasParsingSuccessful;
        do
        {
            Console.WriteLine($"Enter {variableName}");
            var userInput = Console.ReadLine();
            wasParsingSuccessful = double.TryParse(userInput, out result);
            if (!wasParsingSuccessful)
            {
                Console.WriteLine("Invalid format, please try again.");
            }
        }
        while (!wasParsingSuccessful);

        return result;
    }

    public static bool ReadBool(string question)
    {
        bool result = false;
        bool wasParsingSuccessful;
        Console.WriteLine($"{question} Enter Y or N");
        do
        {
            var userInput = Console.ReadLine();
            if (userInput == "Y")
            {
                result = false;
                wasParsingSuccessful = true;
            }
            else if (userInput == "N")
            {
                result = true;
                wasParsingSuccessful = true;
            }
            else
            {
                Console.WriteLine("Invalid format, please try again. Enter Y or N");
                wasParsingSuccessful = false;
            }
        }
        while (!wasParsingSuccessful);

        return result;
    }
}
```

As you can see the only purpose of this class is to provide functions for reading from the console.

Now, the MathUtilities:

```
static class MathUtilities
{
    public static QuadraticFunctionRoots CalculateQuadraticFunctionRoots(
        double a, double b, double c)
    {
        var delta = b * b - 4 * a * c;
        if (delta > 0)
        {
            var firstRoot = (-b - Math.Sqrt(delta)) / (2 * a);
            var secondRoot = (-b + Math.Sqrt(delta)) / (2 * a);
            return new QuadraticFunctionRoots(firstRoot, secondRoot);
        }
        else if (delta == 0)
        {
            var onlyRoot = -b / (2 * a);
            return new QuadraticFunctionRoots(onlyRoot);
        }
        else
        {
            return new QuadraticFunctionRoots();
        }
    }
}
```

And finally QuadraticFunctionRoots class:

```
class QuadraticFunctionRoots
{
    public bool IsNone { get; }
    public bool IsOne { get; }
    public bool AreTwo { get; }

    public double FirstRoot
    {
        get
        {
            if (IsNone)
            {
                throw new InvalidOperationException(
                    "There is zero quadratic function roots.");
            }
            return _firstRoot;
        }
    }

    public double SecondRoot
    {
        get
        {
            if (IsNone)
            {
                throw new InvalidOperationException(
                    "There is zero quadratic function roots.");
            }
            if (IsOne)
            {
                throw new InvalidOperationException(
                    "There is one quadratic function root.");
            }
            return _secondRoot;
        }
    }
    private readonly double _secondRoot;

    public QuadraticFunctionRoots()
    {
        IsNone = true;
    }

    public QuadraticFunctionRoots(double onlyRoot)
    {
        IsOne = true;
        _firstRoot = onlyRoot;
    }

    public QuadraticFunctionRoots(double firstRoot, double secondRoot)
    {
        AreTwo = true;
        _firstRoot = firstRoot;
        _secondRoot = secondRoot;
    }
}
```

This class can hold zero, one, or two results within. It exposes FirstRoot and SecondRoot properties which throw exceptions if an invalid operation is executed,

for example, if the programmer tries to read the second root of a result that only has one root.

So let's summarize what happened: the code got divided into smaller, more cohesive classes. There are no code repetitions now, naming is better and the code style is consistent. No comments are added since no comments are needed - the code speaks for itself.

Is this code perfect? Of course not! No code is perfect, that's for sure. I see couple more things that could be improved, which I skipped for brevity's sake:

- ConsoleReader should not be a static class, it should rather implement an interface like IUserInputReader - this way we could easily change the implementation to, for example, reading user input from a form in a desktop application. Also, this class has a very specific implementation that would probably have to be generalized when more methods for reading various types would be added.
- Also, writing to console should not be done with Console.WriteLine in the QuadraticFunctionRootsCalculator - this way this class is tightly coupled with the Console class. What if we decide to change the way of communicating with the user? This should be, again, some interface like "IUserMessageWriter".

You might argue that this code is actually **longer** than it was before refactoring. Is it a bad thing? No, it isn't! Code refactoring naturally leads to creating more classes, methods, and data structures. Also, please note that I added a brand new entity for representing the roots of the quadratic function - the QuadraticFunctionRoots. It allowed me to have clean, well named and simple-to-use operations on the not-so-simple result of quadratic function roots calculation. In a real-life project, we could have thousands of places where quadratic functions would be used, and having a well-defined class to represent its result would be critical.

Long code is not an issue. Tangled, messy, unreadable, and complicated code is. Don't be afraid to make your code longer during the refactoring. You will still spend less time working on this code than you would spend working on code that is shorter, but uglier.

Tip: other interview questions on this topic:

- "**What is the "ravioli code"?**"
  The "ravioli code" is a term that describes code that contains classes that are easy to understand in isolation, but interactions between them and the project as a whole are not.

- "**What is the "lasagna code?**"
  The "lasagna code" is a code whose layers are complicated and so are the interactions between them. Making a change in one layer heavily affects other layers.

41. What is the Singleton design pattern?

Brief summary: Singleton is a class that only allows creating a single instance of itself, and exposes simple access to that instance.

Singleton is a class that only allows creating a **single** instance of itself, and exposes simple access to that instance.

First, let's see a code that uses Singleton, and then we will see how Singleton may be implemented in C#.

```
var singleton = Singleton.Instance;
var nextSingleton = Singleton.Instance;
Console.WriteLine(singleton.Id);
Console.WriteLine(nextSingleton.Id);
```

The Id is assigned a random value when the constructor of the Singleton class is called. If singleton and nextSingleton variables have the same id, it would prove that they are indeed the same instance. Let's see the console output:

```
singleton: 149c42eb-c9e7-47bf-be72-71d480569212
nextSingleton: 149c42eb-c9e7-47bf-be72-71d480569212
```

It seems like they are the same instance.

Let's see how the Singleton can be implemented in C#. The key is to have a private constructor, to make sure it is not used anywhere outside Singleton class:

```
public sealed class Singleton
{
    private static Singleton instance = null;
    public string Id { get; } //only for presentation purposes

    private Singleton()
    {
        Id = Guid.NewGuid().ToString();
    }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

As you can see, the Instance static property creates an actual Singleton object if it is null, and if it is not - it returns the value of the private field. This way we ensure that at most one instance will be used.

Important note: please be aware that this is the simplest possible implementation of the Singleton class. It is not thread-safe (which means, it may not work as expected if the application uses more than one thread). For details of implementing the thread-safe Singleton I recommend reading this article by Jon Skeet: https://csharpindepth.com/articles/singleton

The most common scenarios of using Singleton are cases when we need a single object shared by the whole application to be responsible for some job. For example, it could be some kind of a logger, which writes application logs to a file. We don't want to have multiple objects that all try to access the same file, because that might create problems if, for example, they tried to write to this file at the same time.

Now when you understand what Singleton is and how it can be useful, I might slightly surprise you by saying that Singleton is widely considered an **antipattern**. Please do not use Singleton.

To understand why that is so, you must understand **what a global state is and why it is a bad thing**. I will not go into details about the global state, because it could easily be a topic for another lecture.

In short, the global state is, as the name suggests, globally accessible throughout the entire application. That means every class may access it, use it, and possibly modify it. It is a very bad thing. It's important that all dependencies of a class are clearly known (we usually make that happen by having all dependencies listed as private fields of **interface** types in a class). The global state is a "hidden" dependency - any class may depend on it. Also, since any class can access the global state, we never know what the global state actually is at the given moment - because any class could have modified it. Next thing is that modifications of the classes that manage global state are very risky - they might affect any place in the application and we would not know.

There are more reasons why the global state should be avoided. If you are curious I highly recommend this article by Miško Hevery: http://misko.hevery.com/code-reviewers-guide/flaw-brittle-global-state-singletons/

Using Singletons means having a global state. Singleton's instance is publically available and any class can use it. That's why Singleton is considered an antipattern.

So what about those cases when we considered Singleton useful - like in the case of having a single logger for the whole application? Well, we can still have **one** instance of a class without using the Singleton design pattern (notice the capital "S"). We should simply have an "application singleton", which is a single object that is created (usually) at the beginning of the program execution, that is then passed explicitly as a dependency to all classes that need it.

Let's see this in a simple piece of code:

```
static void Main(string[] args)
{
    var singleLoggerPerWholeApplication = new Logger();
    var database = new Database(singleLoggerPerWholeApplication);
    var networkConnector = new NetworkConnector(singleLoggerPerWholeApplication);
    var interfaceHandler = new InterfaceHandler(singleLoggerPerWholeApplication);

    RunApplication(database, networkConnector, interfaceHandler);
}
```

Tip: other interview questions on this topic:

- "**What is a global state?**" *It is any state that is reachable from any point of the application. For example, a public field in a public, static class.*

- "**Why is Singleton considered an antipattern?**" *Because it is a piece of the global state, and the global state is hard to control.*

- "**What is the difference between the Singleton Design Pattern and the application singleton?**" *An application singleton is simply a single object used in various places in the application. It does not enforce its "singletoness", unlike the Singleton design pattern. Singleton is a class that only allows creating a single instance of itself, and exposes simple access to that instance.*

42. What is the Builder design pattern?

Brief summary: Builder is a design pattern that allows the step-by-step construction of complex objects.

Builder is a design pattern that allows the step-by-step construction of complex objects.

There are a couple of possible implementations of this pattern. For brevity, I will discuss only one of them, possibly the most popular one.

Let's see a very simple Builder implementation example:

```
public class Person
{
    private string Name { get; }
    private string LastName { get; }
    private int YearOfBirth { get; }

    private Person(string name, int yearOfBirth)
    {
        Name = name;
        YearOfBirth = yearOfBirth;
    }

    public class Builder
    {
        private string _name;
        private int _yearOfBirth;

        public Builder WithName(string name)
        {
            _name = name;
            return this;
        }

        public Builder WithYearOfBirth(int yearOfBirth)
        {
            _yearOfBirth = yearOfBirth;
            return this;
        }

        public Person Build()
        {
            return new Person(_name, _yearOfBirth);
        }
    }
}
```

In this code we have a Builder that will build objects of the Person class. The Builder is a **nested** class in the Person class. Person class has only a private constructor, so we will not be able to use it from the outside. The PersonBuilder will be the only way to construct Person objects. Let's see this code in use:

```
var person = new Person.Builder().WithName("Alex").WithYearOfBirth(1980).Build();
```

So far you may consider Builder an unnecessary complication of the code. Why not simply use the constructor?

Using the Builder gives us a couple of benefits over using the constructor:

- When using the constructor, we need to have all the parameters ready at this particular moment. When using Builder, we can provide them one by one, whenever convenient:

```
static void Main(string[] args)
{
    var builder = new Person.Builder();
    var name = ReadName();
    builder = builder.WithName(name);
    //some other operations
    var year = ReadYear();
    builder = builder.WithYearOfBirth(year);
    //some other operations
    var person = builder.Build();

    Console.ReadKey();
}

private static string ReadName()
{
    //imagine this is some complicated process, like reading from a remote database
    return "Alex";
}

private static int ReadYear()
{
    //imagine this is some complicated process, like reading from a remote database
    return 1980;
}
```

- It is considered good practice to keep the constructor as simple as possible. If we want some more complicated logic than just assigning parameters to values (for example validation, type change, or some calculation of a final value) we may consider having a Builder as the class responsible for all those things:

```
public Builder WithYearOfBirth(int yearOfBirth)
{
    if(yearOfBirth > DateTime.Now.Year || yearOfBirth < 1900)
    {
        throw new Exception("Invalid year of birth");
    }
    _age = DateTime.Now.Year - yearOfBirth;
    _yearOfBirth = yearOfBirth;
    return this;
}
```

- Build methods can be well-named. Look at the code below. Which is more readable - the constructor or the Builder?

```
var dog1 = new Pet("Dog", "Louis Charles Bryan the Third", "Rex", "Tina", "Lucky", "Happy Paws");
var dog2 = new Pet.Builder()
    .WithType("Dog")
    .WithNickname("Rex")
    .WithOfficialName("Louis Charles Bryan the Third")
    .WithFatherName("Lucky")
    .WithMotherName("Tina")
    .WithBreedingCompanyName("Happy Paws");
```

Of course one can argue that we may use object initializer instead of Builder:

```
var dog3 = new Pet
{
    Type = "Dog",
    OfficialName = "Louis Charles Bryan the Third",
    Nickname = "Rex",
    MotherName = "Tina",
    FatherName = "Lucky",
    BreedingCompanyName = "Happy Paws"
};
```

…but it would force us to add public setters to those properties, and we often want to avoid that.

Having a Builder is not always a perfect solution, though. Let's see what are the downsides of Builders:

- We need to write **a lot of extra code**, most of it being a duplicate of the class we add the Builder to. Any change in the properties of that class will force us to also change the Builder
- Constructor forces us to provide all the parameters. With Builder, it is easy to make a mistake of **forgetting to set some of the properties**:

```
var invalidDog = new Pet.Builder().Build();
```

Such an object will have all properties set to null. It doesn't seem like a valid object.

- We can also make some mistakes like **overwriting a property**:

```
var catBuilder = new Pet.Builder()
    .WithType("Cat")
    .WithNickname("Leon");

//some logic here...
catBuilder = catBuilder.WithType("Dog");
var cat = catBuilder.Build();
```

In general - Builder gives us a lot of flexibility, but with that comes a bigger chance of doing something wrong.

As a final note let me mention **records**. Starting with C# 9 we can declare a type as a record (before we had classes and structs only). A record provides a couple of interesting features, one of them being a built-in Builder mechanism implemented by the "with" keyword:

```
var car = new Car() { Brand = "Mazda", Color = "Red" };
var carAfterPainting = car with { Color = "Blue" };
```

Using the with keyword actually creates **a copy** of the original car object, with Color property changed to the new value.

Tip: other interview questions on this topic:

- "**What are the benefits of using the Builder design pattern?**"
  It allows building objects step-by-step. It helps to enclose complex building logic in a separate class. It improves readability, especially when the constructor takes many parameters.

- "**What are the downsides of using the Builder pattern?**"
  It requires a lot of extra code which is at least partially duplicated with the code of the class we build. It causes a risk of omitting some required building parameters. Also, it creates a risk of setting some property twice by mistake, thus overwriting it.

- "**What does the <u>with</u> keyword do?**"
  *It is used to create a copy of a **record** with some particular field or fields set to new values.*

43. What is the Adapter design pattern?

Brief summary: The Adapter is a design pattern that allows converting an interface of a class to the interface expected by a client.

The **Adapter** is a design pattern that allows converting an interface of a class to the interface expected by a client.

Let's see a real-life example to understand it better. Imagine you are working on an application that allows the user to book hotels. The frontend team tells you the user will enter the name of the city, and a list of hotels from this city should be shown on the screen. Together you design the following interface:

```
public interface IHotelsByCityFinder
{
    IEnumerable<Hotel> FindByCity(string city);
}
```

That looks good. The problem is, your team does not implement the search engine used for finding hotels - some other team does that, and you only use the package they publish. The only class from this package that you might find useful looks like this:

```
public class HotelsByZipCodeFinder : IHotelsByZipCodeFinder
{
    public IEnumerable<Hotel> FindByZipCode(string zipCode)
    {
        switch(zipCode)
        {
            case "E1 6AN":
                return new[] { new Hotel("Imperial Hotel"), new Hotel("Golden Duck Hotel") };
            case "E1 7AA":
                return new[] { new Hotel("Ambassador Hotel") };
            default:
                return Enumerable.Empty<Hotel>();
        }
    }
}
```

As you can see, this class finds hotels, but for a given **zip code**, not a city name. Our client (in this case the frontend team) expects an interface that will take the city name. But this isn't such a big problem - given a city name, you can find all zip codes that belong to this city, and then use the HotelsByZipCodeFinder to find

hotels for each of the zip codes. You will need to create an **Adapter** - a class that implements the interface expected by the client, using a class that has a different interface.

Let's see how this could be done:

```
public class HotelsByCityFinderAdapter : IHotelsByCityFinder
{
    private readonly IHotelsByZipCodeFinder _hotelsByZipCodeFinder;

    public HotelsByCityFinderAdapter(IHotelsByZipCodeFinder hotelsByZipCodeFinder)
    {
        _hotelsByZipCodeFinder = hotelsByZipCodeFinder;
    }

    public IEnumerable<Hotel> FindByCity(string city)
    {
        var zipCodes = GetZipCodesForCity(city);
        return zipCodes.SelectMany(zipCode => _hotelsByZipCodeFinder.FindByZipCode(zipCode));
    }

    private IEnumerable<string> GetZipCodesForCity(string city)
    {
        if(city == "London")
        {
            return new[] { "E1 6AN", "E1 7AA" };
        }
        throw new Exception("Unknown city");
    }
}
```

As you can see, the **Adapter implements the interface that is expected by the client** - IHotelsByCityFinder. It has a private field of type IHotelsByZipCodeFinder - that was the type that did **not** meet the client's expectations. The Adapter uses this field to find hotels by zip code. It also has its own way of finding zip codes by the city name (in a real-life project it would be a job of another class, but I kept it in the Adapter for simplicity). This way the HotelsByCityFinderAdapter class adapted the "wrong" IHotelsByZipCodeFinder interface to the "right" IHotelsByCityFinder interface.

This design pattern is commonly used, especially in applications that rely on many external services and work as a "middleman" between them, passing the requests back and forth from one service to another.

Tip: other interview questions on this topic:

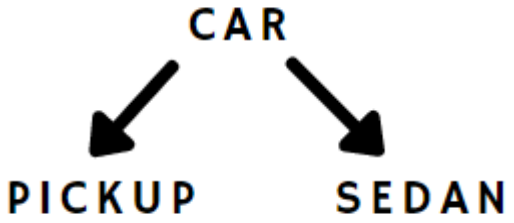
- "**What design pattern would you use if you had some interface incompatible with your needs, and you would like to adjust it**?"
  The Adapter design pattern, as it allows converting an interface of a class to the interface expected by a client.

44. What is the Bridge design pattern?

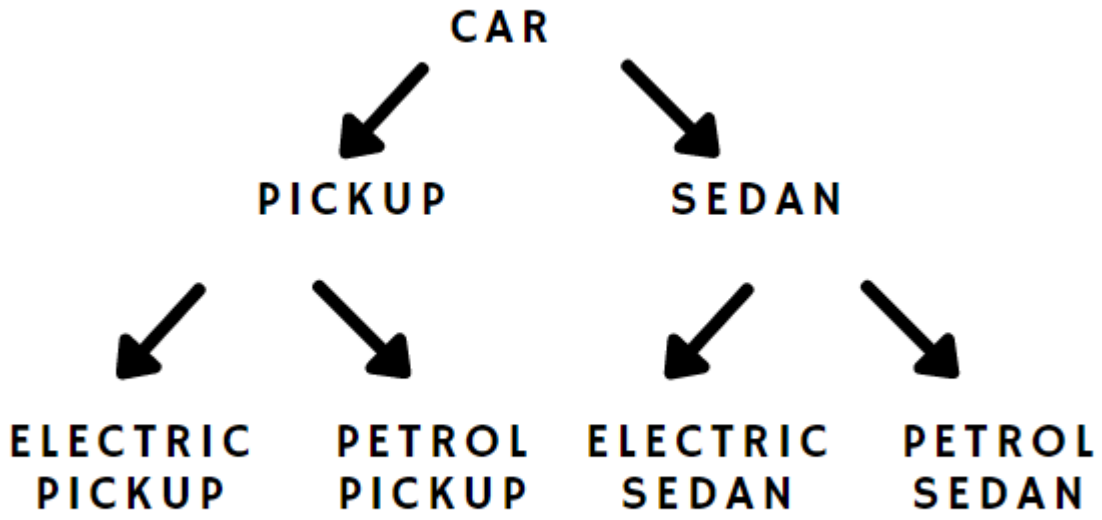
Brief summary: The Bridge design pattern allows us to split an inheritance hierarchy into a set of hierarchies. It is the implementation of the "composition over inheritance" principle.

The **Bridge** design pattern allows us to split an inheritance hierarchy into a set of hierarchies, which can then be developed separately from each other. It is the implementation of the "composition over inheritance" principle, which states that it is better to introduce new features to a class by extending what this class **contains**, instead of extending the **inheritance hierarchy**.

This all sounds a bit complicated, but let's consider a really simple example: we have a base class Car with two inheritors: Pickup and Sedan.

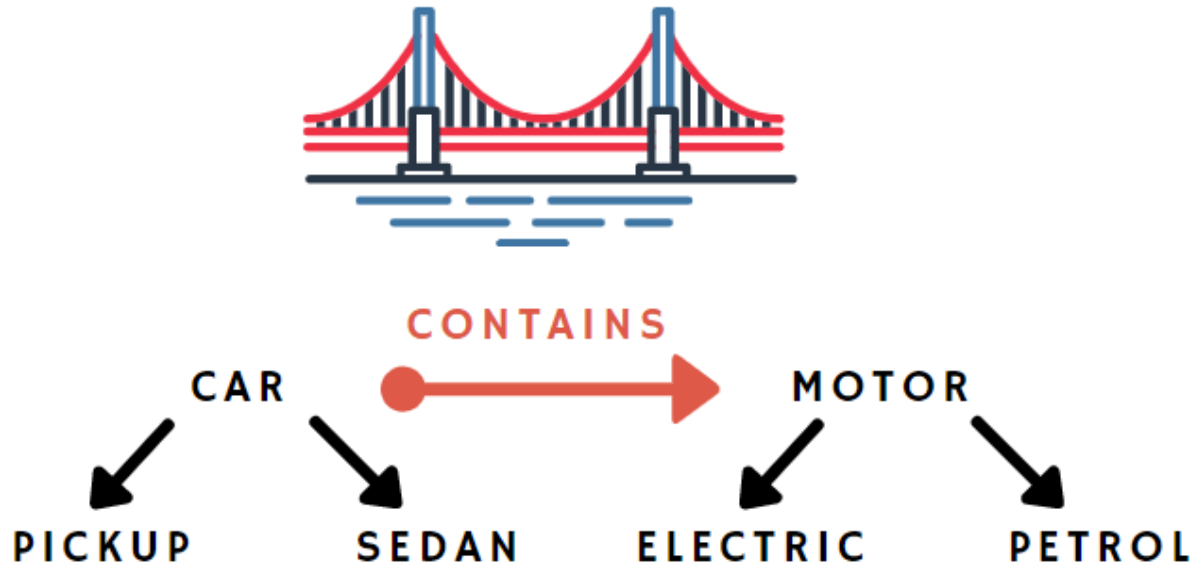


That looks very simple for now. But then the application you develop grows, and there is a need to distinguish electric cars from petrol cars. Let's see how this could affect the inheritance hierarchy:



The hierarchy grew a lot. We suddenly have **seven** classes instead of the **three** we had before. What if we are asked to add another trait that characterizes a car, like manual and automatic gear? We would have to create classes like ManualElectricPickup and AutomaticElectricPickup, and then ManualPetrolPickup and AutomaticPetrolPickup, and so on, and so forth. That would be an explosion of classes, completely unmanageable.

What is the alternative? Well, we can use the **Bridge design pattern**. Instead of expressing a trait of a Car as another layer in the inheritance hierarchy, we **split** the inheritance hierarchy in two. We simply add a new class - like Motor - and then add two inheritors - ElectricMotor and PetrolMotor. Then, the Car class will **contain** a Motor object within. Let's see it in a diagram:



Now, the Car and the Motor are separate entities. We can extend them to our needs, without affecting one another. Also, adding another entity to the picture (like ManualGear or AutomaticGear) is not a problem at all - we will simply add another class hierarchy that represents them.

Let's see this in the C# code. First, the classes representing cars:

```
class Car
{
    public Motor Motor { get; }
    public Gear Gear { get; }

    public Car(Motor motor, Gear gear)
    {
        Motor = motor;
        Gear = gear;
    }
}

class Pickup : Car
{
    public Pickup(Motor motor, Gear gear) : base(motor, gear)
    {
    }
}

class Sedan : Car
{
    public Sedan(Motor motor, Gear gear) : base(motor, gear)
    {
    }
}
```

Now, Gear and Motor:

```
class Motor { }
class ElectricMotor : Motor { }
class PetrolMotor : Motor { }

class Gear { }
class ManualGear : Gear { }
class AutomaticGear : Gear { }
```

Now it should be simple to create any car we want - for example, an electric pickup with manual gear or a petrol sedan with automatic gear:

```
var electricPickupWithManualGear = new Pickup(new ElectricMotor(), new ManualGear());
var petrolSedanWithAutomaticGear = new Sedan(new PetrolMotor(), new AutomaticGear());
```

Thanks to the Bridge pattern, our inheritance hierarchy is kept simple and clean. We won't have any problem with adding new characteristics to the Car class. We can work on each of the families of classes without affecting the other.

Tip: other interview questions on this topic:

- "**What is "composition over inheritance"?**" *It is a principle that states that it is better to design polymorphic and reusable code by using composition rather than inheritance.*

`

45. What is the Factory Method design pattern?

Brief summary: Factory Method design pattern allows us to define an interface for creating objects of a general base type, without specifying what subtype exactly will be created.

The **Factory Method** design pattern allows us to define an interface for creating objects of a general base type, without specifying what subtype exactly will be created.

Let's consider a real-life example first. Nowadays, many companies do not deal with finding new employees by themselves - they rely on hiring agencies. A company knows what skills are expected of the new employee, but besides that, they don't know or care who exactly the employee will be. Let's say the company looks for people with the following skill sets:

```
var firstEmployeeRequiredSkills = new[] { Skills.Jenkins, Skills.Docker, Skills.TeamCity };
var secondEmployeeSkills = new[] { Skills.CSharp, Skills.CleanCode };
var thirdEmployeeSkills = new[] { Skills.Selenium, Skills.Postman, Skills.BlazeMeter};
```

Instead of hiring the people by themselves, the company pays the hiring agency to deal with the entire process - finding candidates, interviewing them, and assessing if their skills match what is expected of them. The company only wants to be "given" an employee, and it doesn't care about details. The interface of the hiring agency that is expected by the company could look like this:

```
interface IHiringAgency
{
    IEmployee Hire(Skills[] expectedSkills);
}
```

As you can see, the hiring agency will "produce" a generic employee. The details of this employee are irrelevant to the client as long as he or she has the required set of skills. It is up to the hiring agency to deal with all the details irrelevant for the client:

`

```
class EmployeeFactory : IHiringAgency
{
    public IEmployee Hire(Skills[] expectedSkills)
    {
        if (Enumerable.SequenceEqual(expectedSkills,
            new[] { Skills.Jenkins, Skills.Docker, Skills.TeamCity }))
        {
            return new DevOps();
        }
        if (Enumerable.SequenceEqual(expectedSkills,
            new[] { Skills.CSharp, Skills.CleanCode }))
        {
            return new CSharpDeveloper();
        }
        if (Enumerable.SequenceEqual(expectedSkills,
            new[] { Skills.Selenium, Skills.Postman, Skills.BlazeMeter }))
        {
            return new Tester();
        }
        throw new ArgumentException("Unexpected skillset");

    }
}
```

The Hire method is an example of a Factory Method. Its only purpose is to create objects of a given base type (in this case, IEmployee). It hides the specific type (DevOps, C# Developer, or Tester) from the client.

Let's see the benefits of using the Factory Method design pattern:

- It separates the creation of the objects from using the objects, making the logic easier to modify independently from the rest of the code when a new type of object is introduced.
- It makes the code less coupled by removing the dependency between classes that need to use IEmployees, and classes that actually implement the IEmployee interface. In our example, any class using the HiringAgency would not be aware of the existence of classes like DevOps or Tester - it would only rely on the abstract type of IEmployee. You can learn why this is important from the "What is the "D" in the SOLID principles?" lecture.
- It helps to eliminate code repetitions - now, by using the EmployeeFactory class, we have one, clear place where the logic of Employee creation basing on the skillset is defined. To learn more about the importance of avoiding repetitions in the code please refer to the "What is the DRY principle?" lecture.

Before we wrap up, let me mention one more thing. There is **another** design pattern that is often confused with the Factory Method design pattern, and it is

`

called the "Static Factory Method design pattern". Similar as they may sound, their purpose is quite different. The Static Factory Method is used as an alternative to a public constructor, mostly used to improve the readability. Let's say you browse through code and you see the following lines:

```
var bankAccount = new BankAccount(true);
var otherBankAccount = new BankAccount(false);
```

It's rather hard to say what exactly "true" and "false" mean here. Let's see this code after it has been refactored to use the Static Factory Method pattern, and then we will see the implementation itself:

```
var bankAccount = BankAccount.ForChildren();
var otherBankAccount = BankAccount.Regular();
```

`

Now it's much better. The first bank account is a special variant designated for children. Let's see the BankAccount class now:

```
class BankAccount
{
    private int _maxWithdrawalSum;

    private BankAccount(bool isForChildren)
    {
        if (isForChildren)
        {
            _maxWithdrawalSum = 1000;
        }
        else
        {
            _maxWithdrawalSum = 10000;
        }
    }

    public static BankAccount ForChildren()
    {
        return new BankAccount(true);
    }

    public static BankAccount Regular()
    {
        return new BankAccount(false);
    }
}
```

As you can see, the constructor of this class is private, so the instances of this class can only be created with the special, well-named static methods ForChildren and Regular.

Tip: other interview questions on this topic:

- "**What is the Static Factory Method design pattern?**"
  Static Factory Method is a method used to create objects. It is used as an alternative to a public constructor, mostly used to improve the readability.

`

- "**What are the benefits of using the Factory Method design pattern?**"
  Separation of concerns (in this case, separating the process of creating the object from using the object), avoiding code duplications and reducing coupling.

- "**What is the separation of concerns?**"
  Separation of concerns is a design principle for separating the code into distinct sections such that each section addresses a separate concern.

`

46. What is the "S" in the SOLID principles?

Brief summary: "S" in the SOLID principles stands for Single Responsibility Principle (sometimes referred to as the SRP). This principle states that a class should be responsible for only one thing. Sometimes the alternative definition is used: that a class should have no more than one reason to change.

First of all, SOLID is a set of five principles that should be met by well-designed software.

"S" in the SOLID principles stands for Single Responsibility Principle (sometimes referred to as the SRP). This principle states that a class should **be responsible for only one thing**. Sometimes the alternative definition is used: that a class **should have no more than one reason to change**.

`

Let's consider the following class:

```
public class PeopleInformationPrinter
{
    private readonly string _connectionString;
    private readonly string _resultFilePath;

    public PeopleInformationPrinter(string connectionString, string resultFilePath)
    {
        _connectionString = connectionString;
        _resultFilePath = resultFilePath;
    }

    public void Print()
    {
        var people = ReadFromDatabase();
        var text = BuildText(people);
        File.WriteAllText(_resultFilePath, text);
    }
    private IEnumerable<Person> ReadFromDatabase()
    {
        var people = new List<Person>();
        using (var connection = new SqlConnection(_connectionString))
        {
            using (var command = new SqlCommand("select * from People", connection))
            {
                connection.Open();
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        people.Add(
                            new Person(
                                reader["Name"] as string,
                                reader["LastName"] as string,
                                (int)reader["Username"]));
                    }
                }
            }
        }
        return people;
    }

    private string BuildText(IEnumerable<Person> people)
    {
        return string.Join("\n", people.Select(person => person.ToString()));
    }
}
```

Is this class responsible for only one thing? Of course not!

- It is responsible for connecting to the database
- It is responsible for transforming a list of people into text
- It is responsible for writing this text to a text file

`

What reasons to change it may have?

- It may need to change if the data source will change - for example, if we decide to read the information about people from the Excel file rather than a database, or if we switch to another database engine
- It may need to change if the formatting of the text will change
- It may need to change if the way of writing the data will change - for example, we decide to write to a PDF instead of a text file

So this class definitely breaks the Single Responsibility Principle. Let's refactor it. If one class needs to be responsible for one thing only, we probably need some more classes:

1. A class that reads the list of people from a database:

```
public class DatabaseReader : IReader<Person>
{
    private readonly string _connectionString;
    public DatabaseReader(string connectionString)
    {
        _connectionString = connectionString;
    }

    public IEnumerable<Person> Read()
    {
        var people = new List<Person>();
        using (var connection = new SqlConnection(_connectionString))
        {
            using (var command = new SqlCommand("select * from People", connection))
            {
                connection.Open();
                using (SqlDataReader reader = command.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        people.Add(
                            new Person(
                                reader["Name"] as string,
                                reader["LastName"] as string,
                                (int)reader["Username"]));
                    }
                }
            }
        }
        return people;
    }
}
```

This class is responsible only for reading the list of people from the database. The only reason to change it could have is if the way of reading would change - for example, the "Name" column in the database would be changed to "FirstName".

`

2. The class that builds a text from a list of people:

```
public class PeopleTextFormatter : IPeopleTextFormatter
{
    public string BuildText(IEnumerable<Person> people)
    {
        return string.Join("\n", people.Select(person => person.ToString()));
    }
}
```

Again, this class would only have one reason to change - if the way how the text is formatted would be changed, for example if we decided to use ";" instead of a new line as a separator between a particular person's information.

3. The class that writes to a file:

```
public class TextWriter : IWriter
{
    private readonly string _filePath;

    public TextWriter(string resultFilePath)
    {
        _filePath = resultFilePath;
    }

    public void Write(string text)
    {
        File.WriteAllText(_filePath, text);
    }
}
```

The same thing here - this class has only one reason to change, for example if we decided to write to another file format than a text file (in this case it might be a better idea to just create a new implementation of the IWriter interface).

All those classes are small, cohesive, and clean. It is much easier to read them and understand what exactly they do.

Now all that's left is to use those classes in the PeopleInformationPrinter class:

`

```
public class PeopleInformationPrinter
{
    private readonly IReader<Person> _reader;
    private readonly IPeopleTextFormatter _peopleTextFormatter;
    private readonly IWriter _writer;

    public void Print()
    {
        var people = _reader.Read();
        var text = _peopleTextFormatter.BuildText(people);
        _writer.Write(text);
    }
}
```

Some may argue "but hey, this class still does three things! It still reads from the reader, it still asks the TextFormatter to build the text, and it still writes to a Writer!". Well, not exactly. This class **only** orchestrates work of other classes - in this case with IReader, IPeopleTextFormatter, and IWriter interfaces. They do the actual work - connecting to the database, writing to the file - and this class **only** works as their manager. It has only one reason to change - if the flow of this process changes, for example, if there is a new requirement to somehow filter the data after reading it from the database, but before sending it to the TextFormatter.

Why is the Single Responsibility Principle important and why should we care to follow it?

- A class responsible for one thing only is **smaller, more cohesive, and more readable**
- Such code is **reusable** - in the example above, the original class would not be likely to be used in any other context. After the refactoring it is easy to imagine plenty of other usages for the DatabaseReader or the TextWriter
- Such code is much **easier to maintain**, as it is much easier to introduce changes and fixes to a class that only does one thing
- Overall, the development speed will be faster and the number of bugs will be smaller

Let's summarize. "S" in the SOLID principles stands for Single Responsibility Principle (sometimes referred to as the SRP). This principle states that a class should be responsible for only one thing. Sometimes the alternative definition is used: that a class should have no more than one reason to change.

`

Tip: other interview questions on this topic:

- "**How to refactor a class that is known to be breaking the SRP?**"
  One should identify the different responsibilities and move each of them to separate classes. Then the interactions between those classes should be defined, ideally by one class depending on an interface that the other class implements.

`

47. What is the "O" in the SOLID principles?

Brief summary: "O" in the SOLID principles stands for Open-Closed Principle (sometimes referred to as the OCP). This principle states that modules, classes, and functions should be opened for extension, but closed for modification.

First of all, SOLID is a set of five principles that should be met by well-designed software.

"O" in the SOLID principles stands for Open-Closed Principle (sometimes referred to as the OCP). This principle states that modules, classes, and functions should be **opened for extension, but closed for modification**. In other words - we should design the code in a way that if a change is required, we can implement it by adding new code instead of modifying the existing one.

It might be a bit hard to understand what exactly it means from a **practical** point of view, but don't worry - we will take a closer look at this principle in a moment. But first, let's understand what is the reasoning behind this principle, and why it is so important. Have you ever heard the expression "The only constant in life is change"? Those are the words of Ancient Greek philosopher Heraclitus, and they are surprisingly fitting to the modern problem of software development. When designing an application, we must always keep in mind that whatever the business requirements are at the moment, they are very likely to change in the future. When they do, we want to be able to:

- Introduce the changes quickly and easily
- Don't break any existing functionality

Following the Open-Closed Principle helps us to achieve it. According to this rule, the code should be:

- **Opened** - that means, it can be extended so new functionality can be added
- **Closed** - that means, we shouldn't be forced to modify the existing code to introduce this piece of functionality

Why is it so important to avoid modifications of the existing code?

- Firstly, it can lead to bugs. Before introducing a code change we have an application that **works**. Modifying a class can make it no longer true. Of course, we should have tests that ensure that everything works fine, but no tests are perfect.

`

- Secondly, changing the behavior of a class can surprise other developers - what if they need the class the way it was? It might not be such a big problem in small projects, but the bigger the project, the bigger the impact of such change might be (not to mention projects that are publicly accessible and can be used by people all around the world). Whenever possible, we want to keep backward compatibility.

Modifying the existing code is particularly dangerous when it affects base types. It's sometimes very hard to anticipate the impact on all the derived types.

Following the Opened-Closed principle mitigates the risk when introducing new functionality. When no changes are done to the existing code, we are sure it still works. The project is more stable and less error-prone.

All right. I hope you now see the benefits of following the Open-Closed Principle. Let's see an example of this principle being broken, and how such a code can be fixed.

`

```
public class Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }
}

public class Triangle
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }
}

public class AreaCalculator
{
    public double Calculate(Circle circle)
    {
        return Math.PI * circle.Radius * circle.Radius;
    }

    public double Calculate(Triangle triangle)
    {
        return triangle.Base * triangle.Height / 2.0;
    }
}
```

This code looks pretty simple. We have two classes representing shapes and an AreaCalculator class that, well, calculates areas. This design **breaks** the Open-Closed Principle. If you are not sure why, imagine what would happen if there was a new business requirement - for example, to start supporting Rectangles and Squares. We would have to add new Rectangle and Square classes (this is fine - we would be **adding** new classes) but we would also have to **modify** the AreaCalculator class. Each time a new shape is introduced, we would need to modify the AreaCalculator class:

`

```
public class Square
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }
}

public class AreaCalculator
{
    public double Calculate(Circle circle)
    {
        return Math.PI * circle.Radius * circle.Radius;
    }

    public double Calculate(Triangle triangle)
    {
        return triangle.Base * triangle.Height / 2.0;
    }

    public double Calculate(Square square)
    {
        return square.Side * square.Side;
    }
}
```

Let's refactor this code. Instead of having area calculation logic in the AreaCalculator class, let's move it to where it belongs - to each of the shapes. To keep backward compatibility, let's leave the AreaCalculator class, but only as a proxy to call the methods from each of the shape classes:

`

```
public interface IShape
{
    double CalculateArea();
}

public class Circle : IShape
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}
```

```
public class Triangle : IShape
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }

    public double CalculateArea()
    {
        return Base * Height / 2.0;
    }
}
```

`

```
public class Square : IShape
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }

    public double CalculateArea()
    {
        return Side * Side;
    }
}
```

```
public class AreaCalculator
{
    public double Calculate(IShape shape)
    {
        return shape.CalculateArea();
    }
}
```

That looks better. Now, if a new shape is needed, the only thing we will have to do will be adding a new class implementing the IShape interface. The AreaCalculator class will not be affected, as it now depends on an abstract interface instead of a concrete class.

Let's consider one more example, that will point out some of the limitations of the Opened-Closed Principle. Imagine we are creating a system for an ice cream parlor. They sell some kinds of ice cream (like vanilla, chocolate, strawberry). They noticed that a huge amount of time is wasted on clients who can't decide which type of ice cream they want. That's why they asked you to create a mechanism that will randomly pick the ice cream for the client. Let's see this code:

`

```
public enum IceCreamType
{
    Vanilla,
    Chocolate,
    Strawberry
}

public class IceCream
{
    public IceCreamType IceCreamType { get; }
    public string[] Ingredients { get; }

    public IceCream(IceCreamType iceCreamType, string[] ingredients)
    {
        IceCreamType = iceCreamType;
        Ingredients = ingredients;
    }

    public override string ToString()
    {
        return IceCreamType.ToString();
    }
}

public class RandomIceCreamGenerator
{
    private Random _random = new Random();

    public IceCream Generate()
    {
        var randomType = GetRandomIceCreamType();
        switch (randomType)
        {
            case IceCreamType.Vanilla:
                return new IceCream(IceCreamType.Vanilla,
                    new[] { "Cream", "Sugar", "Vanilla" });
            case IceCreamType.Chocolate:
                return new IceCream(IceCreamType.Chocolate,
                    new[] { "Cream", "Sugar", "Chocolate" });
            case IceCreamType.Strawberry:
                return new IceCream(IceCreamType.Strawberry,
                    new[] { "Sugar", "Strawberry", "Coconut Cream" });
            default:
                throw new ArgumentException(
                    $"Invalid type of ice cream: {randomType}");
        }
```

You probably know what the problem is - what if a new type of ice cream is introduced? We will have to modify the RandomIceCreamGenerator. That's not right. You might have also noticed that not only the Open-Closed Principle is violated here, but also the Single Responsibility Principle. This class has two

`

responsibilities - picking a random type of ice cream and creating the ice cream basing on the type. Let's create a factory whose only responsibility will be to create the ice cream. You can learn more about Factory Method design pattern in the "What is the Factory Method design pattern?" lecture.

```
public class IceCreamFactory : IIceCreamFactory
{
    public IceCream Create(IceCreamType iceCreamType)
    {
        switch (iceCreamType)
        {
            case IceCreamType.Vanilla:
                return new IceCream(IceCreamType.Vanilla,
                    new[] { "Cream", "Sugar", "Vanilla" });
            case IceCreamType.Chocolate:
                return new IceCream(IceCreamType.Chocolate,
                    new[] { "Cream", "Sugar", "Chocolate" });
            case IceCreamType.Strawberry:
                return new IceCream(IceCreamType.Strawberry,
                    new[] { "Sugar", "Strawberry", "Coconut Cream" });
            default:
                throw new ArgumentException(
                    $"Invalid type of ice cream: {iceCreamType}");
        }
    }
}
```

`

Let's use it in the RandomIceCreamGenerator:

```
public class RandomIceCreamGenerator
{
    private Random _random = new Random();
    private readonly IIceCreamFactory _iceCreamFactory;

    public RandomIceCreamGenerator(IIceCreamFactory iceCreamFactory)
    {
        _iceCreamFactory = iceCreamFactory;
    }

    public IceCream Generate()
    {
        var randomType = GetRandomIceCreamType();
        return _iceCreamFactory.Create(randomType);
    }

    private IceCreamType GetRandomIceCreamType()
    {
        var values = Enum.GetValues(typeof(IceCreamType));

        return (IceCreamType)values.GetValue(_random.Next(values.Length));
    }
}
```

Great. Now this class will not be affected when a new type of ice cream is introduced… but the IceCreamFactory will be! We will have to add another case to the switch. That's actually one of the limitations we encounter when following the Open-Closed Principle. When adding new classes instead of modifying the existing ones, we still need some kind of toggle mechanism to switch between the original and extended behavior. The best we can do is to keep the code implementing such a toggle mechanism in one place - for example a factory class. This way, such change will be simple and will have a small impact on the application as a whole.

The other limitation is that we can't always predict every possible change, and sometimes we will simply be forced to modify the existing code to make it meet the business requirements. We should try to predict the most likely changes that may be needed, but we can't always do it perfectly. But, trying to predict **everything** is also a bad thing. Preparing the code to be modified in every way possible often leads to overcomplication and introducing premature abstraction. As with all things, moderation is recommended. Sometimes it is better to introduce a small modification in the existing code than to spend days on designing advanced mechanisms and abstractions and find out later that we actually never needed them.

`

Of course, bug fixing is another case when modification of the code is simply needed.

Let's summarize. The Open-Closed Principle states that the code should be opened for extension, but closed for modification. That means, when new business requirements are implemented, we should be able to do so by adding new classes instead of modifying existing ones. Following this principle makes the changes easier to be introduced, and mitigates the risk of causing bugs. There are reasonable scenarios when simple code modification is still required - mostly to implement a toggle mechanism between original and extended classes and bug fixing.

Tip: other interview questions on this topic:

- "**What are the good reasons to modify a class, disregarding the Open-Closed Principle?**"
  Firstly, for bug fixing. Secondly, sometimes sticking to the OCP might be an "overkill" - when it generates huge amounts of super-abstract code that brings more complexity than the OCP reduces.

`

48. What is the "L" in the SOLID principles?

Brief summary: "L" in SOLID principles stands for Liskov Substitution Principle. This principle states that we should be able to use a derived type in place of a base without knowing it, and it should not lead to any unexpected results.

First of all, SOLID is a set of five principles that should be met by well-designed software.

"L" in SOLID principles stands for **Liskov Substitution Principle**. This principle states that we should be able to use a derived type in place of a base type without knowing it, and it should not lead to any unexpected results.

Before we continue on this principle, let's just quickly say that **Barbara Liskov**, the author of this principle, was the second woman in history to be awarded the Turing Award. In case you don't know, the Turing Award is sometimes considered a Nobel Prize in computer science. I'm mentioning this because when I was conducting interviews, the candidates often assumed that Liskov was a man, and I think it makes a better impression to show that you know who actually you are talking about.

All right, let's understand better what exactly this principle means.

Imagine you stand on a tower, with a widely-opened window in front of you. I'm standing next to you and I'm giving you objects that all implement the IBird interface that contains the Fly method. You grab those birds and you throw them out of the window. Everything works fine (they happily fly away) unless I'm giving you an ostrich. Thrown out of the tower, it certainly behaves unexpectedly.

This is the example of the Liskov Substitution Principle being broken. You were given an object that said it implemented some interface, and you expected it to behave according to this interface. You did not care about the specific type of this object. Yet, when used according to the parent type, this object did not work as expected.

How to fix the design in this particular situation? Well, at first the IBird interface is not appropriate. Since you are throwing objects from the tower, you should rather be given objects that implement IFlyable interface. An ostrich would not

`

implement this interface, but for example a kite or a drone would - and they would all work fine if you have thrown them out of the window.

As you may already know, more than one SOLID principle is broken here - this design also violates the Interface Segregation Principle. As a matter of fact, those two principles are closely connected - usually, when we break the Interface Segregation Principle and force a class to implement an interface it should not implement, we are also breaking the Liskov Substitution Principle. We will talk more about the relationship between those principles in the "What is the "I" in the SOLID principles?" lecture.

Now, let's take a look at a couple of examples of the Liskov Substitution Principle being broken (all according to Robert C. Martin).

1. **Runtime type switching**

   Here's a simple hierarchy of interface and two classes:

```
interface IFlyable
{
    void Fly();
}

class Bird : IFlyable
{
    public void Fly()
    {
        Console.WriteLine("Flying using fuel of grain and worms.");
    }

    public void FlapWings()
    {
        Console.WriteLine("Flapping my wings.");
    }
}

class Drone : IFlyable
{
    public void Fly()
    {
        Console.WriteLine("Flying using energy stored in battery.");
    }
}
```

   Now, let's see a method that uses the object of IFlyable interface.

`

```
static void FlyAll(IFlyable[] flyables)
{
    foreach(var flyable in flyables)
    {
        if(flyable is Bird bird)
        {
            bird.FlapWings();
            Console.WriteLine("Special case for a bird.");
        }
        flyable.Fly();
    }
}
```

The FlyAll method does not use the IFlyable interface as it was intended - it "hacks" a different path to handle a specific subtype of this interface. This method must know the details of the types derived from the IFlyable interface. That breaks the LSP since we should be able to use any type implementing IFlyable interface wherever this interface is expected, without knowing what exactly this type is or how it works internally. Please note that this code also breaks the **Open-Closed Principle**, as adding new types implementing the IFlyable interface may result in modifications in the FlyAll method. You can read more on the Open-Closed Principle in the "What is the "O" in the SOLID principles?" lecture.

2. **Precondition is strengthened or weakened by a subtype**

   Let's see another example of the LSP being broken. Imagine there is a bank that offers not only ordinary accounts but also special accounts for children. The difference is that the children's account doesn't allow operating on large sums of money. Let's see this in code.

`

```
class BankAccount
{
    public virtual void WithdrawMoney(int amount)
    {
        if (amount < 10000)
        {
            Console.WriteLine($"Withdrawing money (amount: {amount})");
        }
        else
        {
            Console.WriteLine($"Withdrawing sum of {amount} requires extra authorization.");
        }
    }
}

class ChildBankAccount : BankAccount
{
    public override void WithdrawMoney(int amount)
    {
        if (amount < 1000)
        {
            Console.WriteLine($"Withdrawing money (amount: {amount})");
        }
        else
        {
            Console.WriteLine($"Withdrawing sum of {amount} requires extra authorization.");
        }
    }
}
```

This code also breaks the LSP. The base type declares that any amount below 10000 $ will be successfully withdrawn. The consumer of the BankAccount class has the right to assume that calling the Withdraw method for 9999$ will work fine. But then, if they are provided the subtype (ChildBankAccount) instead of the base type, they will be surprised - the operation will not work as expected. Remember, according to the LSP we should be able to use the child class without being aware that it is not an instance of the base class, and it should still work without surprises.

`

3. Abuse of an interface implementation by a subclass

Let's see yet another example of the LSP being broken:

```
class Plane
{
    protected virtual int MaxFuel => 1000;
    protected virtual int RemainingFuel => 100;

    public virtual float PercentOfRemainingFuel()
    {
        return ((float)RemainingFuel / MaxFuel) * 100;
    }
}

class ToyPlane : Plane
{
    protected override int MaxFuel => 0;
    protected override int RemainingFuel => 0;
}
```

I guess by now you have an idea what may go wrong here:

```
static void Main(string[] args)
{
    var plane = new Plane();
    var toyPlane = new ToyPlane();
    PrintPercentOfRemainingFuel(plane);
    PrintPercentOfRemainingFuel(toyPlane);
    Console.ReadKey();
}

private static void PrintPercentOfRemainingFuel(Plane plane)
{
    Console.WriteLine($"Fuel left: {plane.PercentOfRemainingFuel()}");
}
```

When the PrintPercentOfRemainingFuel method is called on a ToyPlane, we receive the following result:

```
Fuel left: NaN
```

`

> That's definitely unexpected. Due to division by zero on float number we received a NaN (Not a Number) result. The LSP is broken here - the subtype gives an unexpected result when used in a place of a base type.
>
> This is another example showing there is a big overlap between Liskov Substitution Principle and Interface Segregation Principle. In this case, we "forced" a child type to implement an interface it should never implement. It doesn't make any sense for a toy plane to declare how much fuel it has remaining.

Let's summarize. The Liskov Substitution Principle says that wherever the base type is expected, we should be able to safely use the subtype without being aware that this is a subtype. When designing the classes and interfaces hierarchy, we should always ask ourselves if it will be safe and reasonable to use the subtype in place of a base type. Also, when we feel that creating a subtype is awkward - for example, we are forced to provide a stub implementation of some methods - we should always stop there and reconsider the design. It is often tempting to quickly create a subtype without laborious redesigning of the whole class hierarchy, but it is definitely worth it - the bugs caused by breaking the LSP may be hard to spot, and finally we will waste even more time.

Tip: other interview questions on this topic:

- "**Who is the author of the LSP?**"
  It is Barbara Liskov, the second woman in history to be awarded the Turing Award.

`

49. What is the "I" in the SOLID principles?

Brief summary: "I" in SOLID principles stands for the Interface Segregation Principle (sometimes referred to as the ISP). This principle states that the clients of an interface should not be forced to depend on methods they don't use.

First of all, SOLID is a set of five principles that should be met by well-designed software.

"I" in the SOLID principles stands for **Interface Segregation Principle** (sometimes referred to as the ISP). This principle states that the clients of an interface should not be forced to depend on methods they don't use.

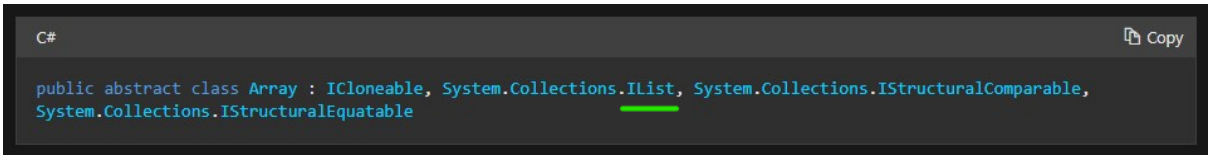
Let's see an example of this principle being violated. What is interesting, this violation was actually done by creators of the C# language themselves! Let's take a look at the code exposing it:

```
class Program
{
    private static void AddElementToList(IList<int> list)
    {
        list.Add(1);
    }

    static void Main(string[] args)
    {
        var list = new List<int>();
        var array = new int[0];

        AddElementToList(list); //this will work fine
        AddElementToList(array); //this will throw NotSupportedException

        Console.ReadKey();
    }
}
```

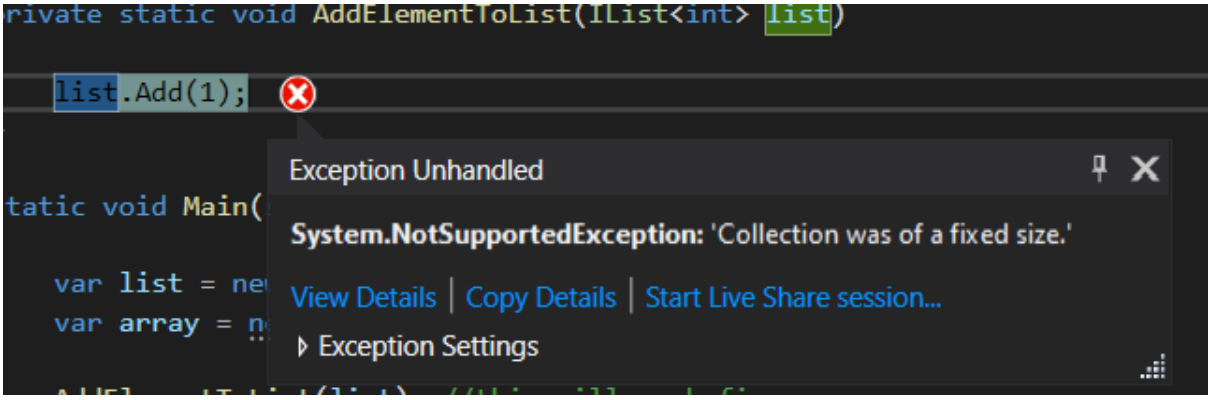
We have a method that takes IList<int> as a parameter and adds an element to it. Note that the Add method is part of the IList<T> interface. Now, what is quite

`

surprising is that in C# the array actually implements the IList interface! Here is a snippet from the C#'s documentation:



Since Array implements this interface, it is perfectly fine to pass it to the AddElementToList method. This method adds an element to the input parameter. But an array in C# is of a fixed size, right? So we can't simply add an element to an array! Let's see what happens when this code runs:



The operation throws an exception! Let's take a look at C#'s array source code, where it "provides" the implementation of the Add method from the IList interface:

```
int IList.Add(Object value)
{
    throw new NotSupportedException(Environment.GetResourceString("NotSupported_FixedSizeCollection"));
}
```

That explains it. The developers of the array class decided it will implement the IList interface, but they obviously were not able to provide a reasonable implementation of the Add method, since an array is of fixed size. So the only thing this method does is throw a NotSupportedException.

This is an example of the Interface Segregation Principle being violated. The Array class is forced to depend on method Add, which it does not need, nor can it provide a reasonable implementation.

How could the developers do it better? First of all, the array, a collection of a fixed size, should not implement any interface that allows operations that modify the collection size. Probably the IList interface should be divided into smaller parts, some of them being responsible for the read-only operations like getting the size

`

of a collection or retrieving the element at index (both array and List could implement such interface), while others providing operations to add or remove elements from the collection (List should implement such interface, but array should not).

Please note that this also breaks the Liskov Substitution Principle. This principle states that we should be able to use a child type in place of a base type safely, without something unexpected happening. But in this case, we can't - if we use the child type Array in place of a base type IList we will encounter an exception when the Add method is called. More on this principle can be read in the "What is the "L" in SOLID principles?" lecture.

Let's see one more example of the ISP being violated:

```
public interface IBike
{
    void Ride();
    void InflateTheTyre();
}

public class Bike : IBike
{
    public void Ride()
    {
        Console.WriteLine("Use your muscles to move forward.");
    }

    public void InflateTheTyre()
    {
        Console.WriteLine("Use the pump to inflate the tyre.");
    }
}
```

This is pretty simple - Bike implements all methods from the IBike interface. Now, sometime later there is a need to represent electric bikes in our project. We add a Charge method to the interface and we create a new derived class:

`

```
public interface IBike
{
    void Ride();
    void InflateTheTyre();
    void Charge();
}

public class ElectricBike : IBike
{
    public void Ride()
    {
        Console.WriteLine("Use your muscles to move forward.");
    }

    public void InflateTheTyre()
    {
        Console.WriteLine("Use the pump to inflate the tyre.");
    }

    public void Charge()
    {
        Console.WriteLine("Charging the battery of an electric bike.");
    }
}
```

The problem is, we need to add some implementation of the Charge method to the Bike class. But a non-electric bike cannot be charged. It seems like we are left but one choice:

```
public class Bike : IBike
{
    public void Charge()
    {
        throw new NotSupportedException("Non-electric bike cannot be charged.");
    }
```

At this moment an alarm should sound in our head. There is something wrong with the interface we declared. Not all bikes are electric bikes. The solution here could be to create a separate IChargable, that could be implemented by the ElectricBike class.

Let's summarize. The Interface Segregation Principle states that clients of an interface should not be forced to depend on methods they don't use. In other words, no class should be forced to implement the methods from the interface

`

that do not fit in this class. When you create a class that implements some interface, and you see that some of the methods that you need to implement are awkward or they just don't fit right, a warning light should start blinking in your head. Do not provide any stub of implementation, hoping no one will ever use it. Spend some time on creating a better design, instead of spending hours in the future finding some tricky bug. Also, be extra careful when adding a new method to an interface. Ask yourself, if any possible subtype of the interface will be needing this method. If not, split the interface.

Lastly, let's try to **define the differences and similarities between the Liskov Substitution Principle and Interface Segregation Principle**. Usually, when we break the ISP, we also break the LSP - implementing a method that doesn't fit in the class will force us to provide a stub implementation, and using this subclass where the base class was expected will give an unexpected result.

In short, the LSP tells us **how** to implement the interface of a base type. The ISP tells us **if** we should implement this interface at all. In practice, refactoring the code to work according to one of them will usually result in also fixing the other one.

Tip: other interview questions on this topic:

- "**What is the difference between the Liskov Substitution Principle and the Interface Segregation Principle?**"
  The LSP tells us how to implement the interface of a base type. The ISP tells us if we should implement this interface at all.

`

50. What is the "D" in the SOLID principles?

Brief summary: "D" in SOLID principles stands for the Dependency Inversion Principle. This principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions.

First of all, SOLID is a set of five principles that should be met by well-designed software.

"D" in SOLID principles stands for **Dependency Inversion Principle**. This principle states that high-level modules should not depend on low-level modules. Both should depend on abstractions.

If this sounds confusing, don't worry. It's actually pretty simple. First, let's consider a real-life example.

Imagine you are an owner of a big online store. You own some office space as well as a huge warehouse where all the goods wait to be sold. Every day there are dozens of new online orders coming in. Your responsibility is to make sure the goods will be delivered to customers. Now, what do you think is a better approach?

- Hire an army of drivers that every day will load trucks of parcels and deliver them all across the country. Also, you will need a smart system for generating address labels as well as picking optimal routes for the drivers. And don't forget to hire some programmers to develop a package tracking system. And also, create some webpage where addresses can report broken or non-delivered packages. Or…
- Hire a delivery company that will do all that.

I guess by now you have an idea of what is simpler and ultimately cheaper.

So, let's say you decided what is better and you signed a contract with a delivery company called FastWheels - in programming language, they will be a dependency you use. Unfortunately, it quickly turns out they are actually not a perfect match for you - they specialize in bicycle delivery, and you mostly sell washing machines. You made a mistake in depending on a **concrete class** rather than an **interface**. So you fire them, and you go to an AnyDelivery company. They are actually not typical - they don't own trucks (or bikes), they don't have warehouses - but they can be a link between you and any delivery company in the world. So you sign a contract

`

with AnyCompany - you specify what you need to be done (washing machines picked up every afternoon, delivered within a 2-days timeframe, etc) and they worry about finding the concrete delivery company that's the best match for your needs.

Now you depend on an **abstraction**, not on a **concrete implementation**. There is a contract that defines what exactly must be done (in C# such a contract is an interface) but from this point you don't care what company delivers the packages (or, speaking in a more programming-focused way, what is the concrete implementation of an interface).

This is the Dependency Inversion Principle in practice. If your company were a class, its responsibility would be to sell goods, not to provide a whole mechanism of delivering packages. It is of course a needed **dependency**, but it should be provided to you via an **interface** (remember, an interface is like a **contract** that defines what operations will be provided by the implementing class). When you depend on abstraction, it is easy to change the concrete implementation to something else when you need to.

Let's try to see it in the code:

```
class YourStore
{
    FastWheelsDelivery delivery = new FastWheelsDelivery();

    public void SellItem(string item, string address)
    {
        var package = PerparePackage(item, address);
        delivery.DeliverPackage(package);
    }

    private string PerparePackage(string item, string address)
    {
        return "package ready to be shipped";
    }
}

internal class FastWheelsDelivery
{
    public void DeliverPackage(string package)
    {
        //delivering accross the country
    }
}
```

`

In the above code class YourStore is tightly coupled with FastWheelsDelivery. You need to modify YourStore class if you decide to use other delivery services. If something changed in the FastWheelsDelivery class, it would most likely affect YourCompany class - it would maybe even force you to change the YourCompany class code.

Let's refactor this code so it meets the Dependency Inversion Principle:

```
class YourStore
{
    private readonly IDelivery _delivery;

    public YourStore(IDelivery delivery) { _delivery = delivery; }

    public void SellItem(string item, string address)
    {
        var package = PerparePackage(item, address);
        _delivery.DeliverPackage(package);
    }

    private string PerparePackage(string item, string address)
    {
        return "package ready to be shipped";
    }
}

public interface IDelivery
{
    void DeliverPackage(string package);
}

internal class FastWheelsDelivery : IDelivery
{
    public void DeliverPackage(string package)
    {
        //delivering with a bike
    }
}

internal class HeavyCargoDelivery : IDelivery
{
    public void DeliverPackage(string package)
    {
        //delivering accross the country even the heaviest washing machines!
    }
}
```

`

Now the YouCompany class depends on an **abstraction** - the IDelivery interface. It doesn't care what class exactly implements it, as long as the job is done. If you don't like how FastWheelsDelivery implements this interface, you can easily swap it for HeavyCargoDelivery, **without making any changes** to YourCompany class.

The Dependency Inversion Principle allows us to achieve more flexible, less coupled code. The classes now live in separation, and only the interface works as a link between them. Remember that concrete classes are modified much more frequently than interfaces, so it's always better to depend on an interface rather than a concrete class. With the Dependency Inversion Principle you can easily swap the implementation of an interface, or even use a mock implementation for testing purposes. The code is easier to maintain and modify.

Tip: other interview questions on this topic:

- "**What's the difference between Dependency Inversion and Dependency Injection?**"
  Dependency Injection is beyond the level of this course, but in short - it's a technique of providing dependencies to a class from the outside (usually with a constructor) instead of creating them within this class. We used it in this lecture when providing IDelivery dependency to YourStore class. To answer the question - Dependency Injection is a mechanism that allows us to provide a dependency, Dependency Inversion is a principle telling that a class should depend on abstraction. Dependency Injection allows us to provide a dependency that was "detached" when following the Dependency Inversion Principle.

FINAL WORD

Thanks for reading this ebook! I hope it will help you during your next interview.

Check out my Udemy courses:

C#/.NET - 50 Essential Interview Questions (Junior Level)

Link: https://bit.ly/3hSRpOq

C#/.NET - 50 Essential Interview Questions (Mid Level)

Link: https://bit.ly/3sC7FsW

LINQ Tutorial: Master the Key C# Library

Link: https://bit.ly/3HqGR33